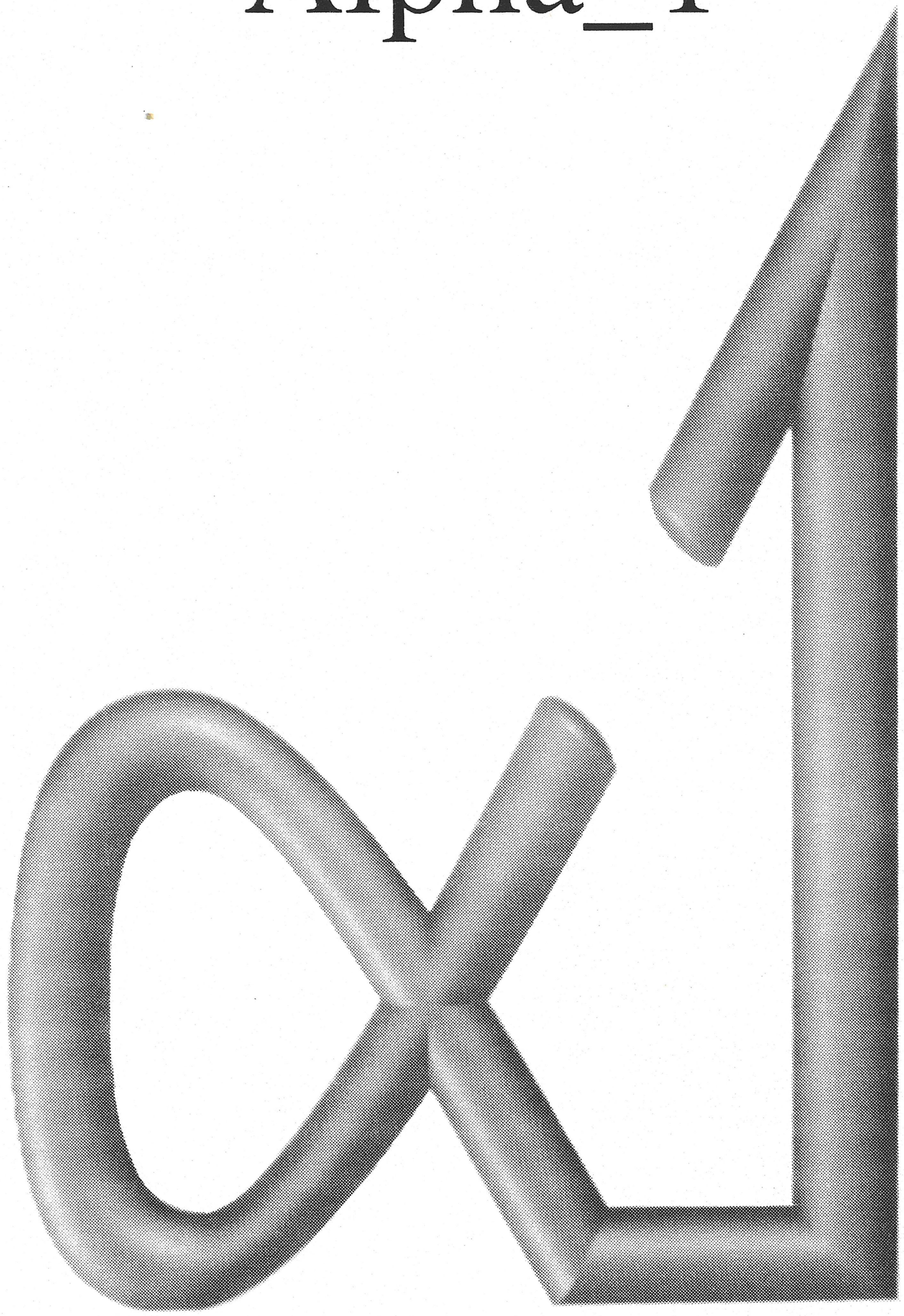
John Peterson

Alpha_1



System Manual

Table of Contents

Introduction

This is the Alpha_1 Programmer's Manual. It contains documentation which is primarily useful for developers of the Alpha_1 system. This manual is aimed at people who will be working on the development effort or working intimately with the system, and will serve as a reference manual for experienced Alpha_1 staff as well as a source of information for new group members. User level information is contained in the companion Alpha_1 User's Manual.

If you are not familiar with the Alpha_1 documentation, see chapter 1 [About the Documentation], page 3 and see chapter 2 [Getting Started], page 5 before going any further. If you are going to be contributing software to the Alpha_1 system, you should also read the section on standards (see chapter 3 [Standards], page 7).

1. About the Documentation

The documentation for Alpha_1 is maintained in an on-line "texinfo" database. From that, an emacs "info" database is derived, which is made up of many small "nodes" linked together in a somewhat hierarchical manner. The printed Alpha_1 manuals are also derived automatically from this on-line database. If you are reading a hard copy at Utah, be aware that it may not be an accurate reflection of the current state of Alpha_1 at Utah, because Alpha_1 is still under development and changes occur often. The on-line version will always be at least as current as any hard copy, and often much more up to date.

This manual is intended to serve as a bridge between the user level functions described in the Alpha_1 User's Manual and the implementation level which is described in comments in Alpha_1 source code. While the details of how many packages are built are not included here, an attempt is made to provide enough information so that the packages can be used in building other parts of Alpha_1.

The Programmer's Manual is less comprehensive (for what it attempts to cover) than is the User's Manual. It is much more difficult to know what things to cover and at what level to describe them. If there are major sections missing, or you wish to make any comments or suggestions, please feel free to submit them.

2. Getting Started

This section contains some pointers for people getting started on the development effort in Alpha_1.

Alpha_1 currently is implemented partly in the C programming language and partly in Lisp. It runs under the UNIX operating system. Some familiarity with UNIX is necessary before any of the Alpha_1 programs can be run. There is a document which gives a beginner's introduction to UNIX in Volume 2 of the Unix user's manual.

In addition, a screen-oriented editor, emacs, is used exclusively for Alpha_1 software development. Anyone who will be doing more than trivial work on Alpha_1 software at Utah will need to be proficient in the use of emacs. Both Unipress emacs and gnu emacs have been used for development, with gnu emacs being preferred currently. Gnu emacs also contains a tutorial for new users. You can run the tutorial by executing the command "control-h t". In this manual (and in the emacs documents), when commands for emacs are described, the notation "M-" means to hold down the *meta* key and type the next character. For example, M-x whould mean type "x" while holding down the meta key. If no meta key is available, the "escape" key can be used instead, but it is not held down. So typing "escape" followed by "x" is equivalent to M-x. The notation "C-" is similar but refers to the *control* key.

Before beginning to experiment with these basic features, new users should request that some Alpha_1 support person provide you with a set of startup initialization files. These should include:

```
.cshrc        # executed each time a c-shell starts up.
.login        # executed at login time by csh.
.emacs_pro    # executed when Unipress emacs starts.
.emacs        # executed when gnu emacs starts.
.a1defs       # loaded by .cshrc, variables for Alpha_1
.aliases      # loaded by .cshrc, common abbreviations
.defs         # loaded by .cshrc, can be personalized
```

The normal way to get this done is to sit down at a terminal and let the Alpha_1 support person set up a directory for you and explain generally what each file does. When these files are correctly in place, you will have an "environment" which matches the one used by the rest of the group. This is a crucial step because "bare" unix and emacs are missing some important utilities which are used constantly and are set up by these files when you log in or start up an emacs. If no system staff is available to help, you can simply do:

```
cd
cp ~alpha1/cmds/.?? .
```

to get the startup files. (Support staff: see section 4.2 [New User Setup], page 43 for a script to help with the setup process.)

Once you have the Alpha_1 environment set up and are familiar with the basics of UNIX and emacs, you will want to get information relating specifically to Alpha_1. The best way to get information about Alpha_1 is to learn to use info in emacs. Info can be invoked as:

```
M-X info
```

and is self-documenting (i.e., there is a tutorial which you can request when info starts up). There are two Alpha_1 menu items in the directory node. One is the User's Manual and the other is this document. If you are reading a paper copy, the information in it is derived from the same text as the info database. The online info databases will always be the most current written documentation on the Alpha_1 system at the development sites.

If you are going to be participating in the development of Alpha_1, you must read the chapter on programming standards (see chapter 3 [Standards], page 7). This chapter describes in detail the conventions and practices which the Alpha_1 group has adopted and is required reading for all members of the group. Software which does not meet these standards will not be incorporated into the system.

If you cannot find enough information for a programming task from the info tree, the actual C and Rlisp code which makes up Alpha_1 is generally very well commented and can be accessed easily with the tags database.

The tags functions in emacs provide methods for accessing portions of the Alpha_1 code directly via keywords. The keywords are currently any function or routine name, any defined constant name, and most of the names of major typedef'd data structures. The command “M-.” will show you the location of the first occurence of a specified tag, and “<*arg*> M-.” will go to the next occurence of the tag. The first time that you invoke “M-.” you will be asked to provide the name of the tags file to be used. The Alpha_1 C code tags are in file “$a/.tags”, and **shape_edit** (Rlisp) tags are in file “$shape/.tags”. The **visit-function** command (bound to “C-z C-v” in gnu emacs or “C-z C-v” in Unipress emacs) works from the same tag table, interpreting the word under the cursor in the buffer as the tag to be visited.

Whenever additions to the Alpha_1 system are made, the author should provide as much documentation as possible. It is absolutely required that all code (including C, Rlisp, and shell scripts) be generously commented and tagged. If a new program or utility has been provided, a document should be prepared describing its use. The document may then be given to an Alpha_1 staff member who maintains the manuals for editing and including into the manual. If you should need to add the information yourself, see section 3.8 [Creating Documentation], page 37. It includes instructions for writing documents according to Alpha_1 style, and for deriving printed and info versions. Generally, though, only one or two people will be maintaining the documents and will have responsibility for installing new documentation. Note that this does not exempt you from writing the documentation for utilities you create in Alpha_1.

1. Installation Guide

This chapter describes the procedures for installing the Alpha_1 Geometric Modelling System under a compatible Unixsystem.

Alpha_1 includes:

- Shape_edit, the interactive model construction environment built on PSL (Portable Standard Lisp). One or more interactive display devices are required to run it. Input to Shape_edit may come from the display device to the Graphical User Interface (GUI) in Shape_edit, or from a command language Programmers User Interface (PUI) embedded in PSL. Use of the PUI requires an editor environment capable of carrying on a dialog with an interactive PSL subprocess. We currently use the GNU and Unipress(Gosling) versions of the emacs editor for this purpose at the University of Utah.
- C utility programs for processing models. Functions include rendering B-spline models into shaded raster images, extracting mass properties of solids, and combining sub-models with boolean set operations (union, intersection, and difference.)
- The Utah Raster Toolkit, a set of libraries and programs used to manipulate and display the RLE (Run Length Encoded) shaded images produced by Alpha_1. Anti-aliased image compositing is featured. (The Utah Raster Toolkit is also available separate from Alpha_1 from the University of Utah Computer Science Department, and in the BRL-CAD distribution.)

Tape Types and Disk Space

We normally send out a machine-independent source tape for Alpha_1, plus machine dependent library tapes and executable binary tapes for each of the machine types you have requested on your configuration questionaire:

source tape
: All sources for Alpha_1, no executables or libraries, machine independent, one copy per site.

library tape
: Libraries and load modules for Alpha_1, machine dependent, one copy per machine type requested.

binary tape
: Executable programs for Alpha_1, machine dependent, one copy per machine type requested.

Each tape contains 15 to 20 megabytes of data, depending on machine type. The size of the data should be marked on each tape. On machine dependent tapes, the version of Unix on which it was generated will also be given.

If we have a compatible Unix version and thus can supply you with a binary tape, you should be able to just read it in and start running Alpha_1. (Some support directories for running Alpha_1, such as command setups ($cmds) and emacs utilities ($aem), are on all of the tapes.)

The library tape contains everything you need to link a set of executables from compiled relocatable files (.o, .a, and .b files.) This may be necessary if we have a Unix host similar to yours, but are not running a compatible version of Unix. Note that 30 to 40 megabytes of free disk space is required, since you will essentially have both a library tape and a binary tape online when you finish. You may then remove the contents of the library tape from disk if desired.

You should be able to rebuild anything on the library or binary tapes from the sources, although the compile time and disk space adds up.

There are sections below which describe the details of installing Alpha_1 from binary, library, and source tapes.

Alpha1 User and Group

Alpha_1 can be installed anywhere in the Unix logical file system. All we ask is that the sources not be publicly readable, i.e., that you restrict the "src" directory ($src) read permissions to the particular user or group licensed for Alpha_1.

You may make the location of the root of the Alpha_1 file tree known as the home directory of an *alpha1* user entered in the host "/etc/passwd" file. This supports one way of getting to Alpha_1 programs and files via references to "~alpha1" in the c-shell, emacs, and PSL. A Unix super-user is normally required to add an entry to the /etc/passwd file.

An alternative is to set the environment variable **alpha1** to the directory where Alpha_1 will be installed, as in **setenv alpha1 /u/alpha1**. This should be done in the c-shell startup files of the Alpha_1 installer and any users who will be using Alpha_1, since $alpha1 will otherwise default to ~alpha1. If you choose this approach, no super-user setup is required.

Installations which intend to use the Alpha_1 source control tools for programming should go further. An *alpha1* user should be set up in "/etc/passwd" to own the sources, and an *alpha1* group should be put in "/etc/group", with Alpha_1 programmers on the group membership list. For example:

```
/etc/passwd:
    alpha1:*:23:6:Alpha 1 Development:/u/alpha1:/bin/csh
/etc/group:
  alpha1:A1:6:alpha1,you,others
```

Reading the Distribution Tapes

We prefer to provide Alpha_1 on 6250 BPI, "tar" format tapes. We can provide 1600 BPI or cartridge tapes if needed. (An extra charge is involved for cartridges.) All that is required to read an Alpha_1 distribution tape is to connect to the (already created) Alpha_1 root directory, mount the tape, and read it in using the Unix **tar** (tape archive) program:

```
tar xp
```

Note the "p" argument to **tar**, which will preserve the permissions of the Alpha_1 files. These permissions do not allow group or public write access.

Binary, library, and source tapes may all be read into the same directory, and will overlay their respective subtrees. Note: The only case in which the order will matter occurs when an Alpha_1 installation rebuilt from a library tape is overlaid with a source tape. The **install-machine** program must then be run to restore the configuration files. (See the section below on rebuilding from a source tape.)

C-shell Environment Setup

You should put ~alpha1/bin (or an explicit path if no user *alpha1* was set up) into your execute search path, the **path** environment variable. This is done in either the ".cshrc" or ".login" file, and should be done by all users wanting to run Alpha_1.

If you have loaded the source tape, it is convenient to refer to subdirectories of the Alpha_1 file tree by symbolic variables which are interpreted by shells, the emacs editor, and PSL. These variables are referenced with the "$" character. (For example, $alb is the same as referring to $alpha1/src/lib/alpha1/bspline.)

The Alpha_1 directory symbols are defined in the "dir.vars" file on directory $alpha1/cmds, which

you should load as part of your c-shell initialization (i.e., you should put a line like **source ~alpha1/cmds/dir.vars** in your ".cshrc" file. The "dir.vars" file sets variables local to the c-shell.

Alternatively, you can load the "dir.env" file (also on directory $alpha1/cmds) in your ".login" file. The "dir.env" file sets environment variables, which are then inherited into other processes. Emacs and PSL both know about environment variables, using the same "$" character for reference.

According to usual Unix practice, the Alpha_1 "bin" directory contains executable programs and scripts. There is also a "cmds" directory, which contains files of c-shell command aliases that provide a more mnemonic interface then the Unix command line arguments for many of the Alpha_1 executable programs. The Alpha_1 Users Manual describes running the programs through the command aliases. It may be useful for frequent Alpha_1 users to load the file "brief.cmds" (from the "cmds" directory) in their ".login" files. The "brief.cmds" file loads rapidly, and defines aliases so that other packages of Alpha_1 command aliases autoload on their first use. The descriptions in the User's Manual do not assume that the autoloading will occur, though.

Installing Alpha_1 from a Library Tape

The Alpha_1 library tape is provided for easy relinking of the Alpha_1 executable programs where operating system incompatibilities (or upgrades) prevent use of the Alpha_1 binary tape for a given machine type. The incompatibilities may be caused by slightly varying Unix kernels due to different NFS (network file system) implementations, different operating system versions (e.g SUN 3.3 versus SUN 4.0) or different, but related, operating systems running on the same type of machine (e.g. BSD Unix versus Ultrix on a VAX or MicroVAX).

The library tape includes all of the ".o", ".a", and ".b" files used in rebuilding a full set of Alpha_1 executables, together with some utility programs which support the rebuilding process.

Most Alpha_1 C programs are linked by the command `cc -o`, and are based on ".o" files in the program directories under $srccmd and ".a" files in the $lib directory. The **shape_edit** program is built differently, using the Unix **ld** command to load ".o" and ".a" files compiled from C together with ".o" and ".b" files compiled from Lisp (PSL).

The companion to the Alpha1_1 library tape is the **install-a1** script. It configures the Alpha_1 system and then relinks the executables from the ".o", ".a" and ".b" files found on the library tape. To use the library tape to install an Alpha_1, do the following:

1. If you have made an account for user *alpha1* on your machine, log in as *alpha1* or use **su alpha1** to run a **csh** as *alpha1*.
2. Make a root directory (usually called "alpha1") if one does not already exist. This should be an empty directory at this point. If you have files under an existing Alpha_1 root directory that you wish to save, you should either copy them somewhere else and clear out the Alpha_1 root directory, or make another (empty) directory into which the library tape can be read.
3. Connect to the empty Alpha_1 root directory and read the library tape with **tar -xp** (or **tar -vxp** for verbose mode).
4. Connect to the "a1make" subdirectory.
5. If you have another installed version of PSL 3.4 you would like to use, you should set the **psl** environment variable to its location "...something.../psl/dist" using **setenv**. Otherwise **install-a1** will reference the PSL files on the library tape.
6. Edit the configuration files if a customized configuration is desired.
7. Run **install-a1**. You may wish to pipe the outout to a log file by running `install-a1 >& log &`.

Interpreting the Log File

If all the executables appear on $alpha1/bin when **install-a1** is done, you should be ready to go. If, however, there are problems you may need to examine the log file and find the problem. This can be somewhat overwhelming to the uninitiated. Here are some hints as to what **install-a1** does and what to look for in the log file when there are problems.

First of all, **install-a1** reports the paths it will use to access the Alpha_1 root directory and root of the psl tree:

```
*** install-a1: Using /u/alpha1/dist/alpha1 as the Alpha_1 root. ***
*** install-a1: Using /u/alpha1/psl/dist as $psl. ***
```

Next **install-a1** runs the **config** script which determines some of the attributes of the Unix under which it is running and proceeds to compile the basic **gnu** and Utah utilities necessary to build the foundations of Alpha_1. The last thing the **config** will do is compile the alpha/src/cmd/util directory, which should look something like this:

```
cd /n/cs/0/usr/alpha1/dist/alpha1/src/cmd/util
gen-makefile -h
make depend
Executing make-depend.
touch depend
make install
... lots of cc lines ...
```

If these compiles all work, **config** has most likely completed without problem. It this directory does not compile for some reason, the rest of **install-a1** will probably produce garbage since the makefiles generated will probably be bogus.

Next **install-a1** runs the **makeall** program to travel over the Alpha_1 tree, generate the makefiles, and relink the executables from the ".o" files and the libraries. The log file will now contain a sequence of sections that say:

```
echo ==== MAKING $alpha1/(directory name) (L) ====
...

echo ==== a1notify ... ====
...
```

This first part (after the "MAKING" line) will have some csh commands in it, and the second section will have the results of the make, if any. There are a number of errors that will occur in this stage that are irrelevant and can be ignored. First, the library tape is a subset of the full Alpha_1 tree and **makeall** will attempt to do some compilation on subdirectories that do not exist. A number of entries in the **makeall** log will look like this:

```
echo ====  MAKING $alpha1/include/check (L) ====
if ( ! -e /n/cs/0/usr/alpha1/dist/alpha1/include/check ) then
echo /n/cs/0/usr/alpha1/dist/alpha1/include/check does not exist.
/n/cs/0/usr/alpha1/dist/alpha1/include/check does not exist.
else
echo
```

These can be ignored. Second, there are some directories with no "makefile.spec" files (from which "makefile" files are generated). These directories are part of the library tape because other directories reference them, but they themselves need no compilation or relinking. These directories are:

```
alpha1/include
```

```
alpha1/urt/lib
alpha1/src/shape/c/sweeps
alpha1/emacs
```

Furthermore, the Utah Raster Toolkit (urt) directories have makefiles, but no "makefile.spec" files. **Makeall** complains about this, but the makes should be successful.

All directories under $alpha1/src where something is relinked (where the real action is) should not have any errors.

Rebuilding Alpha_1 from Sources

Rebuilding a complete Alpha_1 system from sources is definitely not for the faint-hearted. It takes about 60 megabytes of disk, and several hours of compiling. The library tape captures the whole process at a point where it is almost finished, so you are way ahead to load a source tape and library tape together.

On a new machine, you will need a working **PSL** to build **shape_edit**. Unless the machine supports the X window system, you will also need to build at least one "device handler" module.

Since the source tape is machine independent, there is a set of configuration files which must be installed to set up the sources for your machine type. Connect to the $alpha1/a1make/Machines directory, and run the **install-machine** script, giving it the name of your machine type as an argument. (Each machine type corresponds to a subdirectory of the Machines directory.) The configuration files are then copied to the appropriate directories.

Note that if you read in the source tape first and overlay it with a library tape, you can skip the **install-machine** step since the configuration files are included in the library tape.

Also note that if you intend to add to the set of C code linked into **shape_edit**, you must have a **PSL** tape on disk at this point, and set the **psl** environment variable to the full path to the "psl/dist" directory. We only include enough **PSL** with our library tape to link in the existing PSL-to-C interface code.

You may want to edit the configuration files at this point. For instance, to add a PS300 device to a Sun installation, edit the "dev_config.h" on $ai and the "srcdirs" file on $amk to enable the PS300. (The default configurations work for hosts like the Iris where there is no choice of graphics devices, and for other machines supporting the X window system. The only exception to this at present is that the VAX configuration includes a device handler for the ps300, because all our current PS300 installations are connected to VAXes)

To do the actual rebuild, connect to the $alpha1/a1make directory and run **config**. (You probably want to direct all output to a log file and run it in the background, e.g., **`config >& log &`**.) The utility programs will be bootstrapped, all the makefiles regenerated to include the directory paths used in your installation, and a full **makeall** will be done.

You can also rebuild just parts of the system. Use **`config -g`** to go just through the point of regenerating the makefiles, then run **make** on the particular directories you want.

3. Standards

This chapter describes a number of conventions which the Alpha_1 group has adopted concerning coding standards, program construction and system organization. This collection of conventions should be read by all new members of the group before they begin making software contributions to the system, and should probably be reviewed periodically by every group member. The purpose of these standards and conventions is:

1. To achieve some degree of uniformity and coherence in the system as a whole,
2. To make code and programs useable, accessible, and maintainable by others besides the original author, and
3. To attempt to prevent collisions that can easily result since there are usually several group members working on the system simultaneously.

3.1 File System

3.1.1 File Structure

The entirety of Alpha_1 currently resides on the $alpha1 directory. A brief description of the file structure is contained in this section. There are separate subdirectories for

- sources (with program and library subdivisions)
- executables, including scripts needed to work in the system
- commands, definitions and environment setups
- data
- demo programs
- include files
- libraries
- documentation
- special projects

Much of this structure follows the conventions of our environment — namely the way that the Unix system is laid out. Becoming familiar with this structure is essential for those working on the system. The notations in parentheses are the standard Alpha_1 abbreviations for each directory, which are implemented via shell variables in Unix.

At the top-most level is a directory called **alpha1**. It's exact path may differ from one installation to another, but all system references are through the environment variable $alpha1 which contains the particular system path. A short form for users is the shell variable $a.

Subdirectories of the $alpha1 directory and their abbreviations are listed and briefly described below.

cmds	($cmds) Shell alias and variable definitions.
bin	($bin) Executables.
privbin	($privbin) Tools that use special privileges.
lib	($lib) For library archive files.
load	($load) Compiled Rlisp modules.

include	($ai) Alpha_1 include files.
masterfiles	($mstr) Object management in C.
structs	($astr) Object definitions (str files).
src	($src) All sources.
utah	($autah) Utah specific unix utilities.
a1make	($amk) Global make.
docs	($docs) Documentation.
databases	($db) Info and checkout databases.
data	($data) Miscellaneous simple data files.
emacs	($aem) Emacs tags, documents, editor customizations.
projects	($prj) Project development directories, one per project.

The "$src" directory has two major subdirectories: **cmd** and **lib**. The **cmd** directory ("$srccmd") has all the program directories, while the **lib** directory ("$srclib") contains directories for archived libraries of routines.

The major subdirectories of "$srclib" are:

alpha1	($alib) General Alpha_1 library, divided into five sections: support, obj1, obj2, bspline, and mspl.
check	($checklib) Source code management library
intersect	($ilib) Combiner library.
scan	($slib) Rendering library.

The major subdirectories of "$srccmd" are described by the programs they contain. They are:

calc	($calcd) Mass property calculation.
displays	($dispd) **View**, **shade** and other display programs.
hidden	($hidd) Hidden line removal.
intersect	($inter) Combiner
prt	($prtd) Ray caster.
render	($rendd) Rendering programs.
surf	($surfd) Surface maniuplation.
obj-parser	($parsd) Text parser.
structgen	($sgend) Code generators for system objects.
util	($utild) Utility programs.
check	($checkd) Source code management system.

The $shape directory contains all of the Rlisp sources for the interactive geometry editor **shape_edit**. Those sources are divided into several subdirectories as follows:

include	($shi) Definition files for **shape_edit** objects.
model	($shm) Support for modeler objects.
geom	($shg) Geometry routines.
disp	($shd) Display device manager interfaces.
psl	($shp) **Shape_edit** build directory.

applications
(\$sha) Application support (e.g., N/C, FEM)

examples (\$she) Modeling examples.

3.1.2 Access

The Alpha_1 file system is secured via directory protections, controlled by access groups, rather than public-read protecting everything. There are two Alpha_1 groups to which a user may belong: **alpha1** and **reada1**. The **alpha1** group is a subset of the **reada1** group, and consists of just those people who have permission to **write** on the Alpha_1 file system. These are typically people who are developing parts of the system. The **reada1** group contains in addition people who have permission to examine files in the Alpha_1 file system, but who do not need write access to the files.

What follows below is a detailed description of how the protections work, and may not be of interest to the casual reader. The root of all Alpha_1 directories is owned by alpha1, but is publicly readable and executable. The most important consequence of this is that users of Alpha_1 (as opposed to developers) do not have to be members of either the **reada1** or the **alpha1** groups to use the system. All the executables and command files are accessible to anyone. The root of the source tree under the main Alpha_1 root is publically closed, but group readable by **reada1**. All the files under the source tree root are publicly readable, but their paths go through the source tree root so "outsiders" without **reada1** access are unable to get to them or even know of their existence. One can think of the source tree root as a "gate keeper." Everything else under the source tree root is group **alpha1**, so readers, with only group **reada1**, are the ones getting the "public" access on the inside.

Working directores (made by **mknewdir**) are owned by the worker who made them with full access, and are group **alpha1**. By default, these are group **alpha1** writable (by other writers) and publicly read-only (to readers).

Controlled directories (that is, everything "official" in Alpha_1) are owned by **alpha1** with both group and public access limited to reading. But all the Alpha_1 management tools (**checkin**, **checkout**, etc.) run as user **alpha1**, based on the **sa1** command. Only those with group **alpha1** access can run **sa1**, so only writers can install files in Alpha_1 controlled directories.

These are samples of how things should be set up now:

```
# $a - The root of all Alpha_1, owned by reada1.
  drwxr-xr-x alpha1   reada1  /a1/alpha1/

# $src, $rendd - Controlled subdirectories, owned by alpha1.
  drwxr-x--- alpha1   reada1  /a1/alpha1/src/
  drwxrwxr-x alpha1   alpha1  /a1/alpha1/src/cmd/render/
  -rw-r--r-- alpha1   alpha1  /a1/alpha1/src/cmd/render/start.c

# $rendd/rf - Working subdir, made by mknewdir, owned by worker.
  drwxrwxr-x fish     alpha1  /a1/alpha1/src/cmd/render/rf/
  -rw-rw-r-- fish     alpha1  /a1/alpha1/src/cmd/render/rf/start.c
```

Two things that have to be done to maintain this scheme are:

1. When adding users, make sure readers get on group **reada1** only, and writers get on both **reada1** and **alpha1**.
2. Make sure your umask is now 002 rather than 006 as we used to have them. This causes files written under **alpha1** to be readable by readers, rather than publicly read-protected.

3.1.3 Directory Variables

Since the file structure described previously gets fairly deep and complex in places, there is a list of abbreviations for most of the subdirectories. These abbreviations allow easy random access movement throughout the Alpha_1 file system without having to remember or type long pathnames.

The most common use of these is "cd *$abbrev*" to move to the subdirectory whose abbreviation is specified. Directory variables can also be used to easily examine files on other directories without actually changing to that directory, e.g., "ty *$abbrev/file*".

If your ".cshrc" file contains "source ~alpha1/cmds/dir.vars", then both your shell and Unipress emacs will know about the Alpha_1 directory variables without further ado.

There are too many directory abbreviations to list them all here. Refer to "$alpha1/cmds/dir.vars" for a complete and up to date list. The directory variable scheme described above can be cumbersome and is not helpful in gnu emacs, so many programmers are now using an alternative scheme to provide easy access to the Alpha_1 file tree. At Utah, a directory of symbolic links to each directory in the tree is maintained in "/a1/a1_links". The **mk_links** script on $cmds is used to keep it updated with the current version of the "dir.vars" file. Each user makes a symbolic link (called "a") on his home directory to this directory of links:

```
cd
ln -s /a1/a1_links a
```

Then you can find files in emacs via "~/a/*abbrev*" where *abbrev* is the same as the $*abbrev* variable.

3.2 Aliases

Aliases are the "glue" that really connects all the different programs on the Alpha_1 system. Aliases are essentially short abbreviations for stored sequences of commands which may have parameters substituted into them when they are invoked.

The mode variables are the mechanisms by which much of the program-specific "environment" is set up. These actually serve as a more mnemonic means of setting runtime flags which are normally passed in on the command line of the program (instead of having to remember what the "-q" flag means). They also avoid having to set up dynamic program environments in data files.

As an example, there are mode variables which specify for a rendering process the type of shading, the form of shade interpolation, and whether or not transparency is to be used. The alias "render" not only runs the scanline rendering program, but arranges for the data to be converted to binary format, accesses the data on the current data directory, and accesses the current version of the executable file. The "render" alias also arranges for mode variables like the ones described above to be passed into the program whenever it is executed.

Conventions in Defining Alpha_1 Aliases

All of the Alpha_1 program aliases follow a convention which makes it easy to pipe commands together in different ways. The conventions are described in this section in order that other groups of commands will use similar strategies.

Suppose we have a program called **test** for which we wish to create some aliases so that it can be connected with other programs via pipes. Most of the programs that we wish to connect with pipes accept binary object streams as input, so if we wish to run **test** by itself we would first convert the data and then feed that into **test**. We usually create two aliases for **test** as follows:

alias testcmd '$tdir/$tprg'

alias test 'conv <\!:1 | testcmd \!:2'

The first alias references a variable (**tdir**) which holds the name of the directory on which **test** generally resides. It also references another variable (**tprg**) which holds the usual name of the program. Having these defined as variables gives users flexibility in replacing the usual program with a new version which may have a different name and reside on a different directory. The second alias is the one which is invoked when the input to **test** needs to be converted to binary form, as is usually the case when **test** is the first element of a pipe or when it is run by itself. The remaining arguments, after **conv** has taken what it needs, are passed on to **test**. Note that the output redirection is not specified for **test** in either alias. This gives the user maximum flexibility in using it the way he wishes.

These conventions mean that a series of Alpha_1 programs can usually be piped together as follows:

pgmA | pgmBcmd | ... | pgmCcmd >outfile

The first item in the pipe uses the alias which doesn't have the "cmd" on the end; it will invoke **conv** to be sure that the input is in binary form. All the others use the "cmd" form alias because the data is certain to be in binary form since it just came out of another program. The output redirection on the end lets the final results go anywhere the user likes. Since all Alpha_1 programs do their output in binary, if this output redirection is omitted, the last joint in the pipe should be **deconvcmd**. This sends text data to the screen, rather than a binary mess.

For an extensive example of building these aliases, see the "render.cmds" file (on "$cmds") which defines all of the aliases used for rendering.

Brief Commands

The commands for all the Alpha_1 programs take a fair amount of time to load. There is a set of commands which are often included by Alpha_1 programmers in their login sequence, in place of sourcing all the individual command files they might need. The file "$cmds/brief.cmds" provides "temporary" aliases for the commonly used Alpha_1 commands. Invoking one of these temporary aliases causes the full set of commands to be autoloaded. This delays the loading of command aliases and variables until they are actually needed.

There are brief versions of **state, allstate, render, view, conv, deconv** and several others.

The brief command set also defines two useful aliases for loading other commands. One is **getcmds** which unconditionally loads a specified set of commands. This is generally useful from the shell. The other alias is **maybe_load** which conditionally loads a set of commands, depending on whether it has already been loaded. The **maybe_load** alias is used in the other command files to make sure that dependent sets of commands have been loaded, and to avoid resetting them if they have been loaded.

getcmds *package*

Output <none> Load the commands for a particular program.
package <keyword> An identifier for the program. It must be the prefix of a file ending in ".cmds" on the $cmds directory.

maybe_load *package*

Output <none> Conditionally load the commands for a particular program.
package <keyword> An identifier for the program, as for **getcmds**.

Both aliases assume that the requested file has the ".cmds" suffix, so sample uses would be:

```
maybe_load conv
getcmds render
```

Program Directories

The brief command set also contains aliases defining the location of the Alpha_1 program executables which are run by command aliases. (These commands depend on the fact that command aliases follow the conventions stated above.)

Alpha_1 C programs (e.g., **render**) can come from five places:

- working directories (named by user initials)
- installed development directories
- the **bin** directory (default)
- the remote bin directory on a network host machine

The working directories are working spaces for people adding to the system. They have the most current versions of programs under development, but they may not always be stable. The installed development and bin directories contain up-to-date executable versions. The **bin** directory is updated every night from the installed development directories.

You can use one of

demoprogs
binprogs
devprogs

to get the demo, bin, and development versions. The default is currently **binprogs**. Access to versions being tested on new directories is usually arranged by hand by setting the appropriate "*xxx*dir" C-shell variable. You can easily find out the appropriate variable by examining the alias, using

`alias` *xxxcmd*

In addition, **rmtprogs** works by prefixing the directory variables, and thus the command aliases, with a remote machine name. A default host name for remote commands is set in "brief.cmds", and takes effect after the **rmtprogs** command is executed.

rmtprogs

In remote mode, all Alpha_1 main programs such as **conv**, **render**, **view**, and **combine** are run on the remote machine. The **cat** command at the beginning of the pipeline aliases is run on the local machine to read the input data files and pipe them to **conv** and the number crunching processes on the remote host. Commands which produce their output on stdout will work normally, feeding their results back across the network. Commands which write output files, like **render** in **buffer** mode, will write those files on the remote machine filesystem. From there they can be copied back across the network with **rcp** or **cat**. The **remote** command shows the current remote machine name. With an argument, it is like **rmtprogs** but first sets the machine name to its argument. This should typically be one of the machine name aliases for "/usr/ucb/rsh" defined in "/usr/hosts". It is only useful to specify hosts where the "~alpha/bin" directory is installed and where you can log in.

remote
remote *xx*

3.3 Working Procedures

This section describes some practical issues for working in and on Alpha_1.

3.3.1 Working Directories

When you are modifying an existing part of the system, you should **not** be changing the files on the main directories. See the section on make (see section 3.6 [Make], page 22) to learn how to build your own version of Alpha_1 programs for testing of new routines. The current method for developing new routines is to do it on a subdirectory of the directory where the routine normally lives. These subdirectories can be created with the **mknewdir** command described in the chapter on programming tools. This enables other users to access the new versions, but also warns them that they are not necessarily stable yet. The procedures for creating a working directory and installing the completed changes into the system are described in more detail in checkout procedures (see section 3.3.3 [Checkout Procedures], page 13).

3.3.2 Testing

It is important that an addition to Alpha_1 be tested before being installed into the main system. The best mechanism for doing this is to leave the new development on the working subdirectories with easy access for users. Announce its availibility to an appropriate subgroup via the mail system and solicit testing. Example data is helpful in getting other users to run your addition, before they pour their test cases into your code.

Each of the command aliases (e.g., in "render.cmds") has a variable detailing which directory to run the program from. Some also have a variable giving the name of the program. These mechanisms can easily support testing of a new version of a program from a development directory. It is nice to specify how to reconfigure the environment to invoke a test version, when you announce availability for testing. You might even make a C-shell script to do the reconfiguration and one to switch back. This will probably help your testing process as well.

The mechanism of making a new program name on the main directory for your version is intended for programs which will be permanent variants of the command programs, not for incremental test versions. A corresponding responsibility is then incurred by the developer to install permanent command files and/or options to invoke the variants, as with any permanently added part of the system.

Depending on the occasion, it may be appropriate to install the new programs from several days to a week or more later, and clean up the development directories after it is clear that the results are stable.

At all times, the possible negative impact on other users of changes to existing parts of Alpha_1 should be minimized. Try whenever possible to get a global view of what effect your changes will have on the system as a whole. If you have questions, don't hesitate to consult an Alpha_1 staff member.

3.3.3 Checkout Procedures

There exist a set of programs and shell scripts which have been built to insure that only one person modifies any particular Alpha_1 source at one time, and to maintain records of changes made to Alpha_1 source files. These commands manage the state of a simple source librarian database.

In the current system, most Alpha_1 files in the file structure whose root is "~alpha1" are owned by user **alpha1**, and are accorded group read access, but **not** group write access. The **checkout** and **checkin** programs, together with some other related programs, may be used to gain write access to source files. The **checkout** program provides you with a private copy of the source, and maintains a

database which prevents other users from checking the same file out until you have checked it back in. The **checkin** program records the changes you have made, and copies your modified version back as the standard system copy. Your changes and additions to the system are also distributed to other Alpha_1 sites as a side-effect, to keep their copies of the system in sync.

For more detailed information about what these programs do and how they work, see section 4.3 [Check & Friends], page 44. For information about the related programs **chownal**, **sal**, **mkaldir** and **mknewdir**, read the chapter on programming tools.

Below are some recommended procedures for modifying Alpha_1 files.

To edit Alpha_1 source files, you **must** check them out. The recommended procedure is:

1. Connect to the source directory (e.g., "$surfd", "$alib", etc.).
2. Execute **mknewdir**. This creates a subdirectory named with your initials. If you specify an argument, the subdirectory is given that name. The subdirectory will be owned by you.
3. Go to the subdirectory with **cd**.
4. Check out the files you want to edit with the **checkout** program — it is assumed that the files are on the parent directory.
5. Make and test changes on the subdirectory. If the makefile needs to be modified, check out "makefile.spec" as well. Otherwise just run **gen-makefile** and **make-depend** to get one for your subdirectory. If the directory you are working in is a **shape_edit** subdirectory, it is very important that you test the new modules compiled (using build and load) as well as interpretively.
6. Check in the files you checked out and any new files you want to add to the system using **checkin**. They will be moved to the parent directory.
7. Remove the working subdirectory (From the parent directory, `sal rm -rf` will remove the subdirectory and all files in it.)

The recommended method for creating totally new directories (involving a new subdirectory of "$src") is as follows:

1. Follow existing practice in deciding when a new directory is needed and its name and location in $src. Discuss it first with a member of the Alpha_1 support staff.
2. Create the new directory with **mkaldir**.
3. Go to the new directory with **cd** and make a working directory with **mknewdir**.
4. Create your new files and test it there, or move it from elsewhere.
5. Install your new files to the new directory with **checkin**.

Source files (and the makefile) may be checked in individually, or everything on the directory may be checked in at once. For an individual **checkin**, simply go to the working directory and check in the source files with **checkin**.

For directory **checkin**,

1. Check out the parent directory (`checkout -d` ...).
2. Make sure everything you don't want permanently installed in the system is cleaned out. You can check with `textfiles *`.
3. Check in the new directory (`checkin -d` ...). All of the text (source) files on the working directory will be checked in to the parent directory and creation messages will be distributed.

This procedure really only works well for new directories, because you typically want to have reasons associated with each change to existing files, while **checkout** and **checkin** apply the same reason to all changes when done on a directory.

Finally, add your new directory to the "srcdirs" file so **makeall** will run **make** there periodically, keeping it up to date with the rest of the system.

Modifying Header Files

You will discover, if you haven't already, that include files are the mechanism we use to describe common structures and macros that will be used by many files. Include files for Alpha_1 C code are on $ai or its subdirectories, and end in ".h"; while include files for Rlisp code are on $shi and usually end in "-df.r". Makefile dependencies are generated relative to special time files ending in ".t" which correspond to each include file. All files which depend on a particular include file are recompiled when the ".t" files are updated, but not necessarily when the include file itself is updated.

When modifying include files you must decide if the changes you are making are "transparent" or "non-transparent" to existing code in the system. Transparent changes usually occur when you are adding something new to the file (e.g., a new object definition or macro) or only modifying comments or spacing. Transparent changes may sometimes involve changes to existing objects or macros if you are certain that you are simultaneously changing **all** code that references the object or uses the macro.

Non-transparent changes are those which require recompilation of any code that uses the structure or macro you are modifying. Examples include moving a field in a structure or changing the body of a macro.

If the changes you are making are transparent, then the **checkin** procedure outlined above is appropriate. If the changes are non-transparent, then after checkin you will need to cause the time files to be updated with

```
touchal $ai/xxx.t
```

or

```
touchal $load/xxx.t
```

This will trigger recompilation of all files which refer to that include file.

Please consult an Alpha_1 staff member the first time you need to make a non-transparent change, or if you are not sure whether your changes are transparent. In most cases, large chunks of the system will be recompiled when any ".t" file is touched, and this can be very expensive if not kept to a minimum. However, failing to get everything recompiled when necessary is more serious, as inconsistent include definitions may cause subtle bugs to appear almost anywhere in the system.

3.4 C Coding

Some of the coding standards below were first written down in some memos to the group at the end of July, 1980, and some evolved slowly over the next two years. In order for the code in the system to have a high degree of uniformity and readability as a whole, it is essential that these conventions be followed. Further, a certain "style", which is hard to describe as concisely as the conventions below, has been developed in the software which has been written to date. In writing new software, and especially in modifying existing code, the first rule of our coding standards is to **copy style**. Although it is a subtle issue, it makes an important contribution to consistency. If you are modifying an existing module, make your code look like the rest. Even if the code doesn't follow

the guidelines, yours should blend in. Of course, you also have the option of bringing the existing code up to standards, but that should generally be discussed with an Alpha_1 staff member.

It need hardly be said that programs should be written as modularly as possible. In connection with this, many utilities already exist in Alpha_1 (for example, the points library for simple coordinate geometry) in modular form which could be used easily by new programs (see chapter 6 [C Packages], page 79 and see chapter 8 [Rlisp Packages], page 125). People working on the system should be familiar with the utilities that exist so that we don't end up with multiple versions of the same utilities. If you think there might be routines that you could use, don't hesitate to ask! Also make use of the various documentation facilities to locate existing code.

Currently, the text editor of preference for Alpha_1 work is **emacs**. Our version of **emacs** contains a C mode which allows many of the standards below to be followed painlessly. It also has an Rlisp mode for editting Rlisp code. Both of these modes are entered automatically by emacs when you read in a file with the normal C or Rlisp extensions. (These packages were originally written for the Unipress emacs; some of the functions may not have been converted for gnu emacs yet.)

3.4.1 Header Comments

Every file in the Alpha_1 system should contain a standard header. This header is generated automatically with the emacs **make-c-standard-header** command.

M-X make-c-standard-header

It may prompt for a user name and will always prompt for a one-line description of the file. If the environment variable **USERNAME** is set before the emacs is invoked, this value will be used for the user's name in the header. To set up a user name, be sure the following line is in your ".login" file:

setenv USERNAME = 'You R. Name'

The header contains the name of the file, your one-line comment, the author's name and address (as U of U), the date and a copyright notice. It also contains some information for use with a source code control system (RCS) which may eventually be used by Alpha_1. For example:

```
/*
 * sample.c - This is an example file.
 *
 * Author:      Elizabeth S. Cobb
 *              Computer Science Dept.
 *              University of Utah
 * Date:        Wed Apr  6 1983
 * Copyright (c) 1983 Elizabeth S. Cobb
 *
 * $Header:$
 * $Log:$
 */
static char rcs_ident[] = "$Header:$";
```

The header comment may also contain some modification notes at the end. If **major** revisions are made to an existing routine, a note should be added containing the date, the name of the person performing the revisions, and a short (one-line) description of what is being revised. Executing

M-X modification-note

will do this automatically for you. Executing this on the previous example would result in:

```
/*
 * sample.c - This is an example file.
 *
 * Author:      Elizabeth S. Cobb
 *              Computer Science Dept.
 *              University of Utah
 * Date:        Wed Apr  6 1983
 * Copyright (c) 1983 Elizabeth S. Cobb
 *
 * $Header:$
 * $Log:$
 *
 * Modified by: Elizabeth S. Cobb
 *      Date:   Apr  6 1983
 * This is an explanation of why I modified the file.
 */
static char rcs_ident[] = "$Header:$";
```

Modification notes aren't necessary (or wanted) for small fixes and minor additions, but are useful if algorithms have been extensively revised.

3.4.2 Include Files

Most programs will need to include certain header files which define macros and data structures shared by many programs. These header files are in some sense the key to the way we use C in Alpha_1. The Alpha_1 code is really a specialized language of its own, which is defined by the "operators" available through header files. (As examples: list manipulation is in every sense a primitive in Alpha_1 C code, though it is not in the C language; the C memory allocation routine, malloc, is never referenced directly in Alpha_1 C code.)

The include statements for these header files always occur near the beginning of the file, certainly before any code. The emacs command

M-X make-includes

will generate the include format for you, so that you only have to fill in the file names.

3.4.3 Defined Constants

Defined symbols using the "#define" mechanism of the C language should be used wherever possible for parameters such as array sizes. Think carefully before allowing a hard-wired number to appear in your code, especially if the same number appears more than once for the same purpose.

Further, the "#define" entry itself should appear only once. If more than one file uses the parameter, then the "#define" should occur in a header file (".h") which can be shared by the different files.

3.4.4 Naming Conventions

The proper choice of names is exceedingly important in constructing clear, understandable, and maintainable code. Named objects in C include variables, constants, types and structure elements,

functions and macros. In addition, Alpha_1 is built of named files and directories, packages and object types, protocols and operations, programs and commands. We have developed a style of naming a tremendous and continuously growing collection of these things which helps hold the system together and keep it flexible.

The ultimate principle behind our choice of names is to express the meaning attached to names as clearly as possible. Spend a little time thinking about that when adding a new name into the system. Think real hard about it when adding a whole new group of names. When the meaning of a name changes, change the name to reflect it. This is easy to do consistently with global emacs commands like **query-replace** and **grep**.

We tend to use rather long names, consisting of up to several short words or abbreviations connected by the underscore ("_") symbol, which is treated as a letter in C. Related names often share a short prefix indicating the package to which they belong. An example is the "pt_" prefix on the C points package functions (see section 6.4 [Points Package], page 86). Don't get too extreme though. Standard line widths are 80 characters, and a few levels of indentation with a 36-character procedure name and a couple of 20-character variable names runs out of space fast!

The dynamic abbreviations (dabbrevs) package in emacs can be quite useful in keeping name spelling consistent throughout a file. In Unipress emacs, the abbreviation expansion is bound to "M-<space>", while in gnu emacs it is on "M-/". Type the first few characters of a word which appears elsewhere in the file, then execute the expansion function and emacs will find the nearest word that begins with that prefix. It if didn't get the right one, just execute the expansion again and it will look for the next one. The emacs tag functions help keep the global names within the system straight The regular abbreviations package (abbrevs) can be used to expand commonly-used names or patterns.

Case is significant in C names, and is used by convention to distinguish between names handled by the compiler and by the C preprocessor. Constants and macros, both "#defined" in "#include" files and expanded by the preprocessor before the C compiler ever sees them, are normally completely uppercased. Function names use only lowercase letters and underscores, as do names of variables, typedef'ed types, and structure elements.

On occasion, "function-like" macros which do nothing but supply arguments to a support function are lowercased, since it is irrelevant whether they are functions or macros (e.g., **new_knot_vector** in "mesh.h"). Macros which expand to C statements or have their arguments substituted in multiple times are **always** uppercased (e.g., **TRACE_CIRCULAR** and **ADD** in "list.h").

The original definition of the C language stated that only the first 8 characters of a name were significant (7 for external symbols because of the leading underscore). This was all but intolerable as Alpha_1 grew into a medium-large system, contributing to cryptic abbreviations if the names were kept short, or name collisions if longer names were used. When we moved the C code to Berkeley Unix on Vaxen, it became possible to use arbitrarily long names. Alpha_1 development is now dependent on that extension to C, at great benefit to its understandability.

3.4.5 Nesting

Function declarations always begin in column 1. If the function returns a value which is not an integer, the return type declaration should appear on a separate line, and the declarations of parameters should be on the lines following the function declaration:

```
thing *
mk_thg( p1, p2 )
int p1, p2;
```

...

The emacs **make-function** command is helpful for beginning a new routine.

Global declarations also begin in column 1. (It is usually the case that global declarations belong in separate files which are linked in separately, and can thus be accessed by various programs which use common subroutines. As always, use global variables sparingly.)

The minimum indentation nesting is 4 spaces per nesting level. No code should start in column 1. Braces should appear on a line by themselves, in line with the enclosing level, and the statements enclosed by the braces should be indented. The C-mode key "M-" provides this indentation painlessly. The indent program can be invoked with "M-j" to fix up indentation over the whole file. (Warning: copy a ".indent_pro" file from someone before attempting to run the indent program.)

An "else" should be aligned with the corresponding "if"; this includes the "else if" construct:

```
if ( foo )
    ...
else if ( bar )
    ...
else ( something )
    ...
```

In the do-while and repeat-until constructs, the "while" or "until" may appear on the same line with the closing brace.

A null loop statement should appear on a separate line.

Comments and white space should both be used liberally. It is difficult to use too much of either. The C-mode commands for comments are very helpful; documentation can be found in the emacs section of the info-tree.

3.4.6 Major Comments

Every subroutine should contain a major comment (in our "global comment" style) just before its declaration. The comment should begin with a **TAG** entry, which enables the routine to be found with emacs tags, regardless of what directory it resides on or what the external file name is. The comment should have a good description of what the routine is supposed to do. The form which is commonly used is:

```
/*****************************************************************
 * TAG( foo )
 *
 * The foo function does something.
 */
```

The emacs **make-function** command makes it easy to create comments in this style.

3.4.7 Packaging Routines

In the past, the general rule of thumb has been to put every routine in a separate file, with the name of the file identical to the name of the routine. While this is still recommended in many cases, keep an eye out for small "packages" of related routines that can be grouped together into

a single source file. (An example is the symbol table file, "symtab.c", which contains several very short routines for building and accessing symbol tables.) The basic rule is that every routine should be short and comprehensible, and that recompiling a single file should not take very long. In the environment in which we work, the proliferation of files is never a problem, due to the Unix file system and to the existence of **make** to maintain dependency information among many files.

The Alpha_1 libraries generally consist of groups of routines. Each group can be considered a package, even though there is not a formal mechanism in Alpha_1 for specfying what's in a package and what isn't. New additions to the system should either fit into existing packages or create new ones. For information on the contents of existing C packages See chapter 6 [C Packages], page 79.

3.5 Rlisp Coding

Rlisp is the Algol-like (or Pascal-like or C-like) surface language of the Portable Standard Lisp (PSL) system. It has the usual infix arithmetic operators, if-then-else and block structures, and so on. In addition, PSL contains a very powerful set of list and vector datastructure manipulation operations, a resident compiler and an interpreter. There is an extensive manual available on PSL. The Alpha_1 Users Manual contains an appendix which gives a very brief summary of Rlisp syntax which is commonly used in modeling using the **shape_edit** program.

3.5.1 Style for Rlisp

Alpha_1 code for interactive design is available in a "toolbox" environment named **shape_edit**. The **shape_edit** command language is composed of packages of tools "embedded" in the PSL (Portable Standard Lisp) system. Packages for geometry and display are written in Rlisp and compiled, then added into a PSL with Rlisp and saved as the **shape_edit** program.

All that was said about C coding conventions and style applies to Rlisp code developed to become part of the **shape_edit** toolbox. Read it first, if you haven't already (see section 3.4 [C Coding], page 15). Indentation, header comments, and so on all follow the same standards as C, and the same emacs key bindings are used for comments and indentation.

The major comments which precede a function definition in Rlisp are somewhat more important than those in the C code. An automatic help and apropos facility uses the major comments preceding functions to provide information about function actions and calling sequences online during a **shape_edit** session. The guidelines are:

- Comments about the function and its calling sequence should immediately precede the procedure keyword, and describe its calling sequence. The comments should be enclosed in a block of "%"s flush with the left margin. An optional newline between the comment block and the keyword is allowed.
- Extensive algorithm or mathematical descriptions should be separated from the more "helpful" part of the comment by a newline.
- Argument names for a procedure should be descriptive (not mathematical) in nature. For example, use "originalKnotVector" instead of "Tau".

For example:

```
%
% This part of the comment contains all the boring and obscure
% mathematical details about procedure kloo which would be of little
% interest to someone just trying to use the procedure.
```

```
%

%
% This part contains specific information about what procedure kloo
% does, and what the arguments are.  This section shouldn't go on for
% pages and pages: leave very detailed descriptions for user's manual
% or internal documentation.
%
procedure kloo( arg1, arg2, arg3 );
```

Some Rlisp Syntax Notes

The detailed syntax of the Rlisp language is a little different from C.

- Structure -	- Rlisp -	- C -
Comments	% to end-of-line. % another comment.	/* continued across * line boundaries. */
Blocks	... ; or begin ... end;	...
Assignment Op	:=	=
Equality Test	EQ or =	==
Inequality Test	NEQ	!=
Struct reference	Elt Str or Str -> Elt	Str . Elt or Str -> Elt

You also have to be aware that Rlisp statements are **separated** by semicolons, instead of **terminated** by them as in C. This translates to the rule: Don't ever put a semicolon before an "else". Occasionally, you need to leave out semicolons before a "}" or an "end".

Interactive design sessions are conducted with command languages embedded in the Rlisp language, using a more informal style of programming and executing statements one at a time from emacs windows on the terminal for immediate feedback. This interpretive style can include defining interpretive functions or incrementally compiling the functions during a design session, and so it blends finally into the system Rlisp code as new tools and packages are developed and polished.

3.5.2 Naming Conventions in Rlisp

PSL is by default used as a case-insensitive system. All characters of names are uppercased on input, so names match in spite of case differences.

In the Alpha_1 Rlisp code, we have adopted the convention of using long names composed of several short words or abbreviations, as in the C code, but have no need to use case to distinguish between preprocessor names and compiler names. Instead of separating words in a name with underscore characters, the first letter of the words after the first in a name are capitalized, even if it is a one or two character abbeviation, and the rest of the letters in the word are lowercased.

The first letter of a name is lowercased for functions and Rlisp keywords, uppercased for variables. For example,

if, then, else, while, do, repeat, until

are keywords,

crvFromSrf, boolSum, cOrder, arcThru3Pts

are function names, and

DispFile, WindowList, EndPt1, HandleSrf

are variable names.

Sometimes, the first word of a construction function name indicates the return type and later words may indicate the argument types. Following the PSL convention, variables starting with "!*" are (global) boolean flags, and may be manipulated with the **on** and **off** commands, e.g.,

```
on Modeling;
```

(Note that the "!*" is left out in these commands.) Variables ending with "!*" are system control variables with non-boolean values.

3.5.3 Rlisp Definition Files

The Rlisp equivalents of C include files are definition files containing **macro**, **defMacro** and **smacro** definitions, **defSymbols**, **defObjects**, and protocol declarations, and so on. These are all things which **must** be known at compile time, for properly compiled code.

By convention, such files are usually named with a suffix of "-df.r", compiled, declared in the makefile as prerequisites for modules using them, and loaded just like C includes. For example, the line

bothTimes load b!-spline!-df, display!-df;

would go just after the header comment in an Rlisp source file. The **bothTimes** causes the definitions to be loaded at both compile time and load time, giving a consistent environment for the code. Modules are only loaded once, based on the presence of the module name in a variable named "Options!*". Note the "!"s in the load names, which make the dash a part of the name instead of a subtraction operator.

The PSL load library ($pl) contains compiled binary (".b") files for the PSL system, and is initially the only global directory searched by "load" after the current directory. The Alpha_1 load library ("$alpha1/load") is where all the compiled Alpha_1 modules reside, and it is also searched when loading modules into **shape_edit**.

3.6 Make

Every source file in the system should have an entry in an appropriate makefile. There are several important advantages in using makefiles. One is that other users of the system can recompile your program if necessary, without having to know what special libraries it depends on, or which object files need to be linked in. Another, more important, advantage is that makefiles automatically keep track of when modules need to be recompiled, and they can be used to insure that when certain modules change, the programs that depend on them are re-linked.

The importance of makefiles in the way we have been building Alpha_1 should be emphasized. Our success in making the system modular, in using specialized Alpha_1 libraries, in allowing files of concise routines to proliferate, and in taking advantage of incremental compilation is largely due to the fact that makefiles keep all the source code connections straight for us.

It should probably be mentioned here that Alpha_1 makefiles mostly only have local knowlege about dependencies. A makefile describes how to compile and install all of the sources on the directory where it resides. "Makefile variable definitions" which form the basis for dependencies between directories are found in "config.h" files.

There is a "global" makefile which can be used to update all Alpha_1 libraries and programs in the proper order, notifying persons responsible for each directory of the results of the make. This makefile runs automatically every day.

Makefile Portability

An important attribute of our current makefiles is that they are designed to be **portable**. This means that if you want to make modifications to an existing program, you simply carry the modules you wish to modify into a separate workspace (usually a "working subdirectory" of the controlled Alpha_1 directory, see section 3.3.3 [Checkout Procedures], page 13) and generate a makefile for the working subdirectory. Then you can create a new version of the program for testing without impacting the existing system.

A more global version of portability, really "installability", comes from the fact that the makefiles are independent of where the Alpha_1 file tree is rooted in a Unix filesystem. This is no small trick, due to the fact that the make program does not know about "include files" or environment variables such as $alpha1 directly. Without this, we would have to insist that Alpha_1 always be rooted in the same place on each Unix host, e.g., "/u/alpha1".

The portability of our makefiles is based on the fact that they are generated from a more general form: "makefile.spec" files. When makefiles are generated on working subdirectories, symbolic references to compiled files still on the "parent" Alpha_1 directories are automatically inserted. Source files not on the working directory are ignored. Makefile variable values denoting absolute paths are supplied from configuration ".h" files.

It is not necessary to have a "makefile.spec" file on a working directory when making changes which only affect isolated source files. The "makefile.spec" file on the parent directory is used if there is none on the working directory (see section 4.4.1 [Gen-makefile], page 51).

The "depend:" entry of the makefile automatically updates the network of dependencies between the programs, libraries, source and include files you are working on, as files move from the system into your working domain. When the command "make depend" is executed, a program is executed which generates include file dependency declarations for all the sources which are handled by the makefile. The dependencies are appended to the makefile by the program (see section 4.4.4 [Make-depend], page 55 and see section 4.4.5 [Rlisp-make-depend], page 57).

Makefile Types

There are essentially three kinds of C source directories used in Alpha_1, and thus three flavors of C makefiles: program makefiles, library makefiles and include makefiles. Library makefiles build collections of compiled routines which are archived into libraries that are then searched when linking programs. Program makefiles build executable programs, which use the library routines to do their work. Packages in libraries are described to the compiler (and the programmer) by detailed descriptions in ".h" files, which are collected in C include directories for common reference.

There is only one kind of makefile for Rlisp modules which the programmer adding things to Alpha_1 must know about. It compiles modules of Rlisp code into binary files for fast loading, much like C

library makefiles. All of the modules which contain definitions which are loaded at compile time are in a single "shape_edit include" directory which is made first ($shi), and the **shape_edit** executable program which contains all of the Rlisp modules already loaded is in a directory by itself ($shp).

Alpha_1 makefiles generally tend to be clones of existing makefiles or makefile templates. A pleasant side effect is that much of the style of the Alpha_1 makefiles remains constant throughout the system. New makefiles can be generated from the common templates by an interactive Emacs package (see section 3.6.3 [Make-makefile], page 27).

For detailed descriptions of the conventions that are used in Alpha_1 makefiles and the meaning of things to be filled into the makefile templates, see the following subsections.

3.6.1 Makefile Spec File Format

"Makefile.spec" files are translated into normal Unix makefiles by the Alpha_1 **gen-makefile** program. This section describes the basic structure and important fields of the specification files.

"Makefile.spec" files differ from the normal Unix makefile format because they are first passed through the Unix pre-processor to expand configuration variables into values and to conditionally choose sections, and then passed through the **expand-makefile** utility to expand Alpha_1 command templates that require more complicated processing. (See section 4.4.2 [Preprocess], page 52 and see section 4.4.3 [Expand-makefile], page 52 for details).

The Unix pre-processor normally takes lines beginning with the sharp-sign character "#" to be preprocessor commands, which conflicts with the comment convention of makefiles, which use "#"-to-end-of-line comments. Thus, preprocessor command lines in "makefile.spec" files start with a "%" character, e.g.,

```
%if WORK_DIR
%   define UPSTAIRS ../
%endif
```

or

```
%ifdef DEV_XXX
     ...
%endif
```

Makefile variables are as always set by

VAR = *value*

lines in the makefile and referenced by

$(*VAR*)

in other variable lists or makefile targets, prerequisites or actions.

Preprocessor "%define"d constants are referenced without any enclosing syntax,

DEV_*XXX*

so you should keep the distinction in mind when writing or reading "makefile.spec" files.

"Template commands" are expanded by C code in the **expand-makefile** program. They are of the form:

KEYWORD args

and continue across newlines and blank lines without the backslash ("\") at the end of the line that is required to continue normal makefile lines.

Template command keywords currently known are:

CFILES
LFILES
LIB_DIR
LIB_O_DIR
PGM
OPTABLE
LOCPTCLS
LOCOBJS

The **CFILES** and **LFILES** keywords are the only ones you need to know to add new C and Rlisp source files into "makefile.spec" files. The initial argument is the name of the group of source files listed by the rest of the arguments. The **LIB** commands are present as appropriate in the "makefile.spec" templates we use, and the others are used for setting up new programs and their "Object Management" information in the makefile.

3.6.2 Makefile Control

There are some standard features that we put in all "makefile.spec" files, related to the different jobs done by a single "makefile.spec" file. This section describes:

- Standard actions, for the whole directory.
- Variants controlled by conditional logic based on the directory type.
- The detailed control used on working directories to knit together a modified version of parts of Alpha_1 for testing.

Standard Actions

Some standard "makefile targets" are defined in the "makefile.spec" files:

install: This is normally specified in the "$amk/srcdirs" as the action to be taken during the "global make" for library and program home directories. It means that in addition to compilation on the home directory, the binary results will be put in the binary directories $lib and $bin.

default: There is always a "default:" target as the first target in the makefile file, so it is what you get when you run the "make" program without specifying a target. **Default** is also the target specified in "$amk/srcdirs" for the directories without install actions. The default may differ as a "makefile.spec" file is expanded on home, debug, and working directories. In general, the defaults are:

	HOME_DIR	DEBUG_DIR	WORK_DIR
Include dir	all_tfiles	all_tfiles	all_tfiles
Library dir	updatelib	updatelib	updatelib or objs
Program dir	pgms	objs	pgms
Rlisp dir	install	N/A	rebuild

all_tfiles: This causes changes to "time tag" files (".*.t") to be propagated to other ".t" files in an include directory. This is part of the mechanism which causes

libraries and programs to be recompiled throughout Alpha_1 in response to "non-transparent" changes to ".h" files.

updatelib: This is used on library directories and causes compilation of ".c" files into ".o" files, and then the ".o" files to be placed into a load library.

pgms: This is used on program directories, and causes the programs maintained on that directory to be compiled and loaded if something has changed that affects them.

objs: This is used to get just the compilation actions of the **updatelib** and **pgms** targets, without the library maintenance actions on library working directories or on program "debug" directories.

rebuild: This is used for local compilation of Rlisp ".f" files into ".b" files on a working directory, bypassing the installation of the ".b" files into $load which normally happens on Rlisp home directories.

Other actions are used for maintenance purposes:

clean: This is used to remove everything which would have to be recompiled on another machine, in preparation for making an Alpha_1 source distribution.

makefile: The **makefile:** target is used by the "global make" to execute **gen-makefile** if necessary, converting a "makefile.spec" file into a "makefile". This is done when the "makefile.spec" file is modified, or when one of the include files used in processing the "makefile.spec" file changes.

depend: This is used by the global make to execute **make-depend** and/or **rlisp-make-depend** to derive fresh dependency relationships within the makefile when the makefile is regenerated or one of the source files on the directory is changed.

files: This is used by utilities such as **a1count** which have to know about all of the source files on a directory.

Directory Type Variants

There is conditional logic in the "makefile.spec" files which causes them to expand differently on home directories, debug directories, and working directories. "Home directories" are the main Alpha_1 directories where the master copies of all source files live. Home directories are owned by the pseudo-user **alpha1** and are not writable by anyone but **alpha1**. Programs running as **alpha1** are used to maintain the sources and compile the system. (See section 4.3 [Check & Friends], page 44.)

Anyone may put in conditional logic in a "makefile.spec" file, using the **HOME_DIR**, **WORK_DIR**, and **DEBUG_DIR** preprocessor variables in "%if" statements. Examples of this abound in the existing "makefile.spec" files.

On home directories, C code is always compiled with the optimizer flag turned on which disables compilation for source debuggers, so "debug directories" are provided directly under home directories. They are always named "debug" and owned by **alpha1**, and have no source files in them. Instead they compile the C source files on their parent directory into unoptimized ".o" files for the debugger.

"Working directories" (see section 3.3.1 [Working Directories], page 13) are programmer-owned subdirectories of home directories where we develop new parts of Alpha_1 or change old ones. Only the subset of the sources actually being changed need to be on the working directory, with the rest on the home directory.

Working Directory Sections

There are often chains of file dependencies stretching across several working directories belonging to a programmer in different parts of Alpha_1: from include working directories containing modified ".h" files to the library and program directories where the definitions are used in ".c" files; from the library working directory where ".o" files are compiled from ".c" files to the program working directories containing programs that must be linked with the new function definitions for testing; and so on.

Each "makefile.spec" file has a "Working directory section" close to its beginning that allows overriding the standard include file directory, library locations, and so on. This section is contained in an "%ifdef WORK_DIR" that will prevent the working directory definitions from taking effect on **alpha1** directories.

Even on working directories, the lines in the working directory section are normally commented out. For instance:

```
#INEW = -I$(I)/new
```

would be changed by John Q. Programmer in his working directory "makefile.spec" file to:

```
INEW = -I$(I)/jqp
```

to make ".h" files in his "$ai/jqp" subdirectory known to the C compiler.

> Note: Be sure to restore these lines to their original form before checking in a modified "makefile.spec" file from a working directory.

C Libraries are overridden individually from program directories rather than specifying a directory, since all of the libraries are installed in a single directory ($lib). There are two ways of doing this provided:

```
#AL = A1_SRC/lib/alpha1/new/libA1.a
#AL = A1_SRC/lib/alpha1/new/ALL_OFILES $(lib)/libA1.a
```

The first is for use when lots of things are being changed on the library working directory and a library is being built there. The second is for use when only a few ".c" files are being changed on the library working directory and the ".o" files are to just be loaded into the program and the normal library searched for the rest. (An "%if" switch at the beginning of the library "makefile.spec" file determines which treatment the working directory will get.)

All working directory sections also have a section at the end like this one from the $alib directory:

```
# Override default homedir of ".." if on a work dir not just below $alib.
# HOMEDIR = A1_SRC/lib/alpha1
```

We almost never build working directories which are not directly under the home directory, but this is intended to allow for that eventuality. It is used by deleting the "#" before the **HOMEDIR** to override the default homedir of "..", which is the parent directory. (One problem with this on large directories like $alib with lots of files in them is that the full path to the home directory tends to be a much longer string than "../", which can cause the **make** program to overrun its internal buffers and core dump.)

3.6.3 Make-makefile

Make-makefile is an interactive emacs command that constructs "makefile.spec" files for new directories from templates. An autoloaded definition of **make-makefile** is one of the things defined by the "(load "alpha1")" in your ".emacs_pro" or ".emacs" file.

To use it, first create an empty "makefile.spec" file on the appropriate directory with "C-X C-F" or "C-X C-V", then type "M-X make-makefile <CR>". **Make-makefile** prompts for a letter specifying whether to generate a Program, Library, Include, or Rlisp "makefile.spec" file and reads in the appropriate template file from "$bin/lib".

If executed in a buffer which is not empty, **make-makefile** will also prompt whether to wipe out the previous contents. Finally, the interpret-makefile-template command is executed to translate the buffer from a template into a real "makefile.spec" file. (You can execute **interpret-makefile-template** or **interpret-template-region** yourself if it ever becomes necessary to resume interpretation or add a section of template to an existing "makefile.spec" file.)

"Blanks in the template are delimited by braces "{...}" and may be embedded in a single line or span multiple lines. During interactive interpretation, the blanks are filled in using the emacs *minibuffer* prompt window at the bottom of the screen. A prompt string is shown first in the minibuffer to tell you what is expected, followed by a colon and the default value for editing. While prompting for a value, "{{" and "}}" mark the place in the makefile where the pattern will be replaced.

You may freely move out of the minibuffer into other windows to pick up strings such as filenames and return to the minibuffer to insert the value. Carriage return tells emacs you are satisfied with the value, so it goes on to the next one.

Optional and repeated fields have the words "(optional)" and "(repeated)" prefixed on their prompts. Wipe out the default value to give a null response which will quit a repeated item or throw away an optional one. Single-line repeated fields are used to build up lists of files, such as the files in a **CFILES** list. For this reason, the default value of single-line repeated forms is usually a single blank, which is a makefile word separator. To quit a particular list, you must say "<RUBOUT><CR>" to give a null response. Single filenames may be given on each iteration of a list pattern, or many filenames may be given at once.

Multi-line repeated fields are used to build the body of a "makefile.spec" file out of a sequence of sections describing packages of files. Rather than presenting multi-line optional and repeated fields in the minibuffer, the template interpreter first asks whether you desire that section in your "makefile.spec" file. A "y<CR>" response signifies "yes", just a "<cr>" skips that field and goes on.

Fields within an optional or repeated multi-line section form are expanded, and then a *recursive edit* on the section is entered to encourage getting its format and details right while it is still fresh in your mind. Type your favorite way of executing the emacs pop-level command to resume template interpretation.

If you see "(global)" prefixed to the prompt, the replacement you give for the pattern is going to be replaced throughout the rest of the template as well as the place that was presented for editing. This is mostly used for inserting directory names in "makefile.spec" files.

The following sections describe the sequence of blanks in the different template files, together with their uses, prompts, and valid responses. In the descriptions below, prompts and default values will be given first separated by a colon, then descriptions. A default value which is a single space character is denoted by "<BLANK>".

3.6.4 Include Makefile Template for C

There isn't much to do on makefiles for include directories. Their job is to keep the dependency network of ".t" files that echoes the relationship between ".h" files that "#include" other ".h" files. Since those dependencies are automatically produced by **make-depend**, the "makefile.spec" file is largely standard on all subdirectories of $ai.

Description:<BLANK>

Give a few words describing the purpose of the directory.

(global) Directory char:x

The directory symbol for a subdirectory of $ai always starts with "ai". Often, only a single character (at most a short word) comes after the "ai". Change the *x* default to a character or short word arrived at by carefully examining the "$cmds/dir.vars" file.

When you install the corresponding library directory, put that symbol in "$cmds/dir.vars" and also add an "IX" definition (where the *X* is substituted as above) into the "$ai/makefile-defs.h" file.

Definitions directory name:

Type the actual name of the include directory, then "<CR>".

3.6.5 Library Makefile Template for C

(global) Library abbreviation:*Abbr*

This is the abbreviation which will be used with the "-l" flag to the C compiler to load libraries, e.g., "-lA1". It thus tends to be both short and capitalized for Alpha_1 libraries. List the Alpha_1 $lib directory to see which library abbreviations are already used. The abbreviation will be substituted throughout the rest of the file, so when you see *Abbr* in what follows, it really means what is entered to this prompt.

(global) Directory char:*x*

This character is the basis of the Alpha_1 directory symbols destined for "$cmds/dir.vars", e.g., give an "f" for "$flib" and "$aif". It is also used as the basis of the makefile symbols for the subdirectory of $ai corresponding to the library, e.g., "f" causes references to "$aif" and its working directories by the names "$(IF)" and "$(IFNEW)" to appear in makefile **IFLAGS** lists and working directory sections. You should carefully look in "dir.vars" to pick the library directory symbol, which always ends in "lib".

The **IFLAGS** makefile variable holds the **-I** flags telling the C compiler where to find ".h" files. Unlike program directories, we never leave ".h" files on library directories, installing them on $ai or a subdirectory of $ai corresponding to the library. here. The **IFLAGS** list always contains:

- "$(INEW)" which is specified on program and library working directories to tell of a working directory under $ai
- "-I$I" which is $ai itself,
- "$(IXNEW)", a working directory under $aix, and
- "-I$(IX)" (the "x"s have been replaced by the "directory char:" above.)

If this is a new library, you will not yet have put a definition of "$(IX)" in "$ai/makefile-defs.h", so there is a line that prompts

(global) Library directory name:*libname*

and substitutes it into definitions of **IX** and **HOMEDIR**. This reflects the convention that the library directory name and the corresponding subdirectory of $ai have the same name:

#IX = A1_INC/libname
#HOMEDIR = A1_SRC/lib/libname

Packages are named collections of related source files. There is nothing to a library makefile package section except a commented **CFILES** list.

Do you want a (Repeated) Package entry?

Respond "Y<CR>" if you want to enter another package, "<CR>" to quit. You will then be prompted for:

(global) Package name:*pckg*

The package name is used only within the makefile, so just use a short word that will identify the group of files that makes up the package.

Program description:

Here is an opportunity to more fully describe the purpose of the package in a block comment before the **CFILES** list. One line or a few lines are appropriate here. Make sure you put the makefile comment line character "#" on the beginning of any lines after the first.

(repeated) Package C files:<BLANK>

Enter the list of C source files for this package. When you are done entering C file names, you descend into a recursive edit to look it over and correct it with a prompt:

Editing: (repeated) Package entry

On exit from the recursive edit you will be asked if you want any more package entries.

The rest of the makefile is either standard on all library directories or filled in by **expand-makefile** or **make-depend**. Some of the standard lines come from the "$ai/lib-defs.h" include file.

3.6.6 Program Makefile Template for C

Program makefiles by their nature have a lot of options to specify, since they bring together all of the parts of the system used by programs: include directories, libraries, local ".c" and ".h" files, and "object management" features including dispatchers and local object types and operation protocols. This is compounded by the fact that parts of the system may be on a working directory somewhere instead of in the usual places on home directories.

Program makefiles can be pretty simple, but you have to filter out all the advanced capabilities of the "makefile.spec" template which you are not going to use. The best approach on first reading this section is to skip anything having to do with object management and concentrate on getting an overview.

Program description:<BLANK>

Give a few words describing the purpose of the directory.

(global) Directory symbol:**pgmd**

The Alpha_1 directory symbol destined for "$cmds/dir.vars". If not preparing an Alpha_1 permanant directory, just hit <CR> to leave it as "pgmd". Otherwise, carefully look in "dir.vars" to pick the program directory symbol, which always ends in "d".

(repeated) Program name:<BLANK>

The list of programs to be built by this makefile should be listed here. These will each have a corresponding program section below.

(repeated) Installed program name: $(BIN)/

These are the paths to the installed versions of the programs on "~alpha1/bin ($bin)". The same names should be entered here as for the PGMS list above, with "$(BIN)/" prefixed.

The "$(IFLAGS)" makefile variable holds the **-I** flags telling the C compiler where to find ".h" files. The list always starts with:

- "-I." to search the current directory,
- "$(IHOME)" which is automatically defined to access the parent directory when on working directories,
- "$(INEW)" which is specified on program and library working directories to tell of a working directory under $ai, and
- "-I$I" which is $ai itself.

Following those comes an opportunity to specify that include files on a subdirectory of $ai are needed for your program:

Do you want a (optional) Specialized Alpha_1 includes?

If everything you will need is in $ai and $alib, just hit "<CR>" to go on. If you want, for example, includes from $ais and function definitions from $slib, you will need to say "y<CR>" and answer the prompt:

(global) Directory char:*X*

by replacing *X* with an "S" and then hit "<CR>". This will arrange for references to the other include directories.

Since the "Directory char:" prompt is nested in the "Specialized Alpha_1 includes" prompt, you will be placed in a recursive edit since it is slightly more complicated than usual. In this case, there is little chance of error and you probably want to pop right out of the recursive edit with "C-M-z".

(Optional) Device includes: DEV_INCLUDES

If any of your programs will be running display devices, hit "<CR>" to accept this, otherwise erase it to omit.

The **LIBES** list describes the libraries which will be searched when building any programs from this makefile. It always searches "$(AL)", the general Alpha_1 library, after any specialized libraries.

If you specified that you wanted "Specialized Alpha_1 includes", you will get the prompt:

(Optional) Specialized Alpha_1 lib: $(SL)

(Where the "S" should have been replaced by this time by the character you specified to the "Directory char:" prompt above.) Otherwise, you will get the prompt:

Do you want a (Optional) Specialized Alpha_1 lib?

In either case, you want to type a "<CR>" to accept the definition or omit it, respectively.

(repeated) Graphics dev lib: LIB_*xxx*

After the specialized and general Alpha_1 libraries, **LIBES** searches graphics device libraries. The "LIB_*xxx*" symbols are defined in the site-specific file "$ai/dev_config.h" so that they will give access to the installed locations of the display device libraries for any devices present. Devices provided for but not present are no problem since the LIB_*xxx* symbols are always defined but have null values if the display device is not present at an Alpha_1 site.

Omit this if no graphics devices will be used by programs in this directory, and give a null response in any case when they have all been specified. (Note that you may want to specify the device libraries with individual programs in program sections rather than in the **LIBES** list.)

(optional) C Math lib: LIB_M

The C math functions are almost always needed on the **LIBES** list. This probably shouldn't even be an option — just accept it.

(optional) Utah local lib: LIB_LOC

The library containing Utah local routines such as the **scanargs** and **eprintf** functions is also usually necessary. Accept this one too.

If you specified that you wanted "Specialized Alpha_1 includes", you will get the prompt:

(Optional) Workdir specialized includes: #ISNEW = -I$(IS)/new

(Where the "S" should have been replaced by this time by the character you specified to the "Directory char:" prompt above.) Otherwise, you will get the prompt:

Do you want a (Optional) Workdir specialized includes?

In either case, you can use this opportunity to specify where working directory versions of include files are located.

Do you want a (optional) Workdir specialized lib?

If you want to load modules from working directory versions of libraries other than libA1, you can specify those here.

The working directory library definitions are:

#SL = A1_SRC/lib/libdir/new/libAbbr.a
#SL = A1_SRC/lib/libdir/new/ALL_OFILES $(LIB)/libAbbr.a

(Where the "S" in "SL" should have been replaced by this time by the character you specified to the "Directory char:" prompt above.) You then get two prompts:

(global) Library directory name:libdir

and

(global) Library abbreviation:Abbr

to fill in the definitions with the directory name and library abbreviation string. Values entered for the directory name and library abbreviation should be:

Symbol	Directory	Library
SL	scan	SC
IL	intersect	I
TL	trace	Tr
CHECKL	check	Check

For example if you said "S" to the "Directory char:" prompt above, SL is the library symbol corresponding to the path to the libSC.a library, so replace "libdir" with "scan" and "Abbr" with "SC".

Finally you get a recursive edit to confirm the values entered:

Editing: (optional) Workdir specialized lib

Exit from the recursive edit when you are satisfied.

Program directory name:

The directory name where the makefile will be installed should be given here.

Do you want a (Repeated) Program entry?

These are long sections with many options. Respond "y<CR>" if you want to enter another program, "<CR>" to quit. You will then be prompted for:

(global) Program name:pgm

This name becomes the executable name of the program, as well as becoming the unique prefix of all the other symbols defined in the program entry.

Program description:

Here is an opportunity to more fully describe the purpose of the program in a block comment at the beginning of the program entry. One line or a few lines are appropriate here. Make sure you put the makefile comment line character "#" on the beginning of any lines after the first.

(repeated) Program C files:<BLANK>

Enter the list of C source files which make up this program.

(optional) Optable C file: pgm-optbl.c

If you are going to have an object management dispatching table ("OPTABLE") below, the automatically generated source file should be specified here. Usually you just need to change "pgm" to the name of your program (or an abbreviation).

Do you want a (optional) Local protocol dispatcher?

If you are going to have object management operations local to this directory, say "y<CR>" to include this source file, "<CR>" to omit. The default value is

ptcl-disp.c

where the "ptcl" is actually a global abbreviation substituted from the next prompt:

(global) Protocol abbrev:ptcl

You get a recursive edit to confirm you have done the right thing:

Editing: (optional) Local protocol dispatcher

Pop out of it immediately if you like the results, or edit before popping back to the prompting level if you want.

Now we finally get to specifying exactly how the program is to be built. The program name ("pgm") has already been substituted, a reference to the "$(pgmOBJS)" variable is in place, and it prompts for:

(repeated) Additional libraries:<BLANK>

to be searched before the "$(LIBES)" list. Type "<RUBOUT><CR>" if you don't need anything else loaded with this program.

Do you want a (optional) Non-default generic operation dispatch table?

This is the **OPTABLE** section. Just type "<CR>" if you don't need one. Otherwise, say "y<CR>" and you are prompted for **OPTABLE** command arguments:

(optional) Local protocols: -P ptcl

This argument to **OPTABLE** gives the protocol abbreviation, which has already been filled in if you specified a local protocol dispatcher above. Omit if not. Similarly,

Do you want a (optional) Local objects?

specifies that there are object types local to this directory, and if so the default value is:

-O objs

and the object abbreviation is prompted for and substituted:

(global) Object abbrev:objs

Again, you get a recursive edit level in which to confirm the results:

Editing: (optional) Non-default generic operation dispatch table

Pop back to the prompting level to confirm.

Do you want a (optional) Local protocol dispatcher and -ops.h?

You do if you have local operation protocols, otherwise omit. The default is:

LOCPTCLS ptcl

where the "ptcl" has already been substituted in.

Editing: (optional) Local protocol dispatcher and -ops.h

Pop back to the prompting level to confirm.

Do you want a (optional) Local objects -objs.h -objs.mstr?

You want this if you specified local object type definitions, otherwise omit. The default is:

LOCOBJS objs

where the "objs" abbreviation has already been substituted in.

Editing: (optional) Local objects -objs.h -objs.mstr?

Pop back to the prompting level to confirm.

You can share the object management entries among many programs if desired. In this case, you might want to group the optable, dispatchers, and/or operation files in separate **CFILES** lists to ease specifying the corresponding **OBJS** variable in the "Additional libraries:" lists of the other programs.

At the end of the operation table section you get another recursive edit because all of the other questions about local objects, etc. are nested in the dispatch table section.

Editing: (optional) Non-default generic operation dispatch table

Pop back to the prompting level to confirm.

Finally, the end of a program section. You will be put into a recursive edit to look over whole the program section and correct it. On exit from the recursive edit you will be asked if you want any more program entries.

The rest of the makefile is either standard on all program directories or filled in by **expand-makefile** or **make-depend**.

3.6.7 Makefile Template for Rlisp

Rlisp directories are closest in spirit to the C library directories, without all the complication of building a Unix library of the results. **Rlisp-make-depend** generates compilation actions that install the resulting compiled lisp files on the $load directory.

Directory description:<BLANK>

Give a few words describing the purpose of the directory.

(global) Directory char:x

The directory symbol for a subdirectory of $shape always starts with "sh", those under $shg start with "shg", etc. Often, only a single character (at most a short word) comes after the base string. Change the x default to a character or short word arrived at by carefully examining the "$cmds/dir.vars" file.

Do you want a (optional) .t file generators? (Only for lisp include dirs.)?

Probably not. Type "<CR>". The only place this is normally used is on $shi, and you shouldn't be running this function there.

Directory path under $shape:

Type the path of the rlisp directory from $shape, then "<CR>". E.g., the path from $shg is "geom" and from $shgs is "geom/srfops".

Package Sections

Packages are named collections of related source files. There is nothing to an Rlisp makefile package section except a commented **LFILES** list.

Do you want a (Repeated) Package entry?

Respond "y<CR>" if you want to enter another package, "<CR>" to quit. You will then be prompted for:

(global) Package name:pckg

The package name is used only within the makefile, so choose a short name that describes the files that make up this package.

Package description:

Here is an opportunity to more fully describe the purpose of the package in a block comment before the **LFILES** list. One line or a few lines are appropriate here. Make sure you put the makefile comment line character "#" on the beginning of any lines after the first.

(repeated) Pckg lisp source files:<BLANK>

Enter the list of Rlisp source files which make up this package. When you are done entering ".r" file names, you descend into a recursive edit to look it over and correct it with the prompt:

Editing: (repeated) Package entry

On exit from the recursive edit you will be asked if you want any more package entries.

The rest of the makefile is either standard on all Rlisp directories or filled in by **expand-makefile** or **rlisp-make-depend**.

3.7 Command Aliases

The Alpha_1 project has gained immensely from the use of the Unix operating system. The availability of C-shell aliases and shell variables made it unnecessary to write a separate command parser. The text processing tools provided by Unix made building source modification and control tools relatively easy. Parser generation tools were used to create a parser to read data files, and to write programs that automatically generate code from an object description. Pipes made it easy to plumb together separate programs for performing various tasks. And (Gosling) Emacs, with its subprocess control, provided a perfect editing front end for **shape_edit**.

This appendix describes only the use of C-shell aliases and pipes to provide a command language and allow communication between separate programs. The use of other Unix tools in the development of Alpha_1 is described in the Alpha_1 System Programmer's Manual.

The command language that drives all the Alpha_1 programs except **shape_edit** is written as a set of C-shell aliases.

The command for running the **render** program is typical. A top-level command "render" invokes a data conversion program **conv** and pipes the result into the **render** command. The first argument is passed to **conv**, and the rest to the **render** program. Typically, though, only one argument, a list of input files, is supplied, because all the flag arguments are stored in C-shell variables and passed to the **render** program by the alias. Note that both the directory on which the **render** program lives, and the name of the program itself are easily modifiable. This aids greatly in testing new or experimental versions of the program.

```
render =>
        conv !:1 | rendercmd !:2-*
rendercmd =>
        $rndir/$renderprg $shader $hilightflg $shademode $ovlyflg
           $bufflg $cullflg $lightflg  $flipflg $diskzbflg $bkgndflg
           $vwportflg $transpflg $opacityflg $quiltflg $dblresflg
           $visflg $segsflg $aspect $filterflg $clrinterpflg
           $highlightmode
```

The top-level alias is the one which is invoked when the input to "render" needs to be converted to binary form, as is usually the case when "render" is the first element of a pipe or when it is run by itself. These conventions mean that a series of Alpha_1 programs can usually be piped together as follows:

```
pgmA | pgmBcmd | ... | pgmCcmd >outfile
```

The first item in the pipe uses the alias which doesn't have the "cmd" on the end; it will invoke conv to be sure that the input is in binary form. All the others use the "cmd" form alias because the data is certain to be in binary form since it just came out of another program. The output redirection on the end lets the final results go anywhere the user likes. Since all Alpha_1 programs do their output in binary, if this output redirection is omitted, the last joint in the pipe should be "deconvcmd" unless the program does not produce an output stream (and most display programs do not).

The **conv** alias is somewhat interesting; it demonstrates a method of passing several files as one argument, and also of being able to pick up the files from a preselected data directory. The variable "datadir" is set to the name of the directory where the data files are located, with a trailing "/". To get files from the current directory, or from several directories, "datadir" is set to the null string. A list of files separated by commas as the first argument to **conv** will then be properly expanded as arguments to "cat".

```
conv =>
        cat $datadir{!:1} | convcmd !:2-*
```

All flag setting is done through other commands. For example, the aliases below allow the user to set some of the lighting options for the rendering program. The variable names that start with "X" are set to a human readable string describing the value of the particular attribute. The variables without an "X" are set to flag values to be passed to the program. A "state" command echoes the values of the "X" variables, so the user can see the flag settings. The default value echoes nothing, so a "show_defaults" command is provided.

```
blinn     set shader=''; set Xshader='blinn'
cosine    set shader='-C'; set Xshader='cosine'
cornell   set shader='-l'; set Xshader

white     set hilightflg='-w'; set Xhilightflg='white'
nowhite   set hilightflg=''; set Xhilightflg

smooth    set shademode='-g'; set Xshademode='smooth'
flat      set shademode=''; set Xshademode='flat'
normal    set shademode='-n'; set Xshademode
dynamic   set shademode='-N'; set Xshademode='dynamic';
```

3.8 Creating Documentation

This section describes the procedures which are required to add documentation to the Alpha_1 User's Manual or the Alpha_1 System Manual. It describes adding to the online data files formatting that information for printing, and extracting the info database..

In general, only one or two Alpha_1 staff members will be given write access to the manual manuscripts which contain the Alpha_1 manuals. The purpose of restricting the write access of the actual manual text to a small group is to maintain a consistent style, approach and organization to the manual as a whole. This does not relieve the rest of the developers from their task of writing documentation for new parts of the system (or even undocumented old parts). Documents and corrections should be submitted to the editors when new software or changes are installed. If these documents meet the standards of the manual, they will be installed in the manual. If they do not meet the standards, they will be returned to the author with a list of requested improvements and revisions. The editors do not assume responsibility for writing the entire manual, only for insuring that it is maintained according to a set of agreed-upon standards.

The manuals are currently written in **texinfo**, a set of macros for use in the **TeX** typesetting system. **Texinfo** provides a facility for deriving both a printed hard copy versions and online **info** versions for use in **emacs**.

Before adding to the manual, read the **emacs** information available in the **info** tree which describes the **texinfo** commands. The Alpha_1 manuals are structured as described in the texinfo documentation, so follow those guidelines. The major rules for formatting Alpha_1 texinfo documents concern font selection and the formatting of function declarations. (There are a number of emacs functions which can be used in texinfo mode when adding documentation. Ask one of the editors or look on $aem for the file.) As for writing code in Alpha_1, perhaps the most important convention of all is to copy the style of the existing work, making yours blend in.

Texinfo allows four fonts: times roman (@r), italics (really slant @i), bold face (@b), and typewriter (@t) fonts. The only fixed width font is typewriter. The body of the manual should always be set in times roman. Since this is the default font, you usually don't need to think about this. Variables or values which are meant to be substituted (such as function arguments) should be set in italics. Also, technical terms are italicized when they are defined. The names of functions, programs, global variables, global constants, and Lisp identifiers should be set in bold face. If the name of a program or function is used repeatedly in a small region of text (such as a manual page for that program) the first use should be in bold face and the rest can be times roman. Finally, text which is to be typed verbatim by the user should be set in typewriter font.

Formatting Function Calls

The format for function declarations is as follows:

```
@quotation
@findex function-name
@b{function-name}( @i{arg-1}, @i{arg-2}, ..., @i{arg-n} )
@table @i
@itemx Returns
<datatype> A summary of the function.
@itemx arg-1
<datatypel | dataype2> Meaning of arg-1.
@itemx arg-2
<returns datatype> Meaning of arg-2.
@itemx arg-n
<opt returns datatype> Meaning of arg-n.
@end table
@end quotation
```

This results in a formatted document which looks like this:

function-name(*arg-1*, *arg-2*, ..., *arg-n*)

Returns	<datatype> A summary of the function.
arg-1	<datatypel | datatype2> Meaning of arg-1.
arg-2	<returns datatype> Meaning of arg-2.
arg-n	<opt returns datatype> Meaning of arg-n.

The function name is set in bold face followed by the argument list. Each argument is set in italics. Note the parentheses and commas are not included in any font selectors. The argument list is formatted in an @table environment with each argument set in italics (indicated by the @i after the table command). The first item in the table is the return type of the function, the word "return" is capitalized. The datatype returned by the function is the first item in the function summary. If the function has no return value Nil is listed. Following the return type is a short summary of the actions performed by the function. The rest of the argument list follows the return type. Each argument is listed in an @itemx command with the datatype of the argument and its meaning following. If an argument can be one of several datatypes each datatype is listed separated by vertical bars. If an argument is modified by the function call the word "returns" is placed before the datatype. If an argument is optional the abbreviation "opt" is placed before both the datatype and any returns keyword. If a function accepts an arbitrary number of aruments (all of the same type), use "@i{Argl}, @dots{}" to denote this. The @itemx line should also contain "Argl, @dots{}".

Formatting Global Variables and Constants

Global constants also have a special declaration format. It is

```
@quotation
@vindex variable-name
@b{variable-name}
@table @i
@item <datatype>
If true, summary of the variable (default value).
@end table
@end quotation
```

This results in a formated display which looks like

variable-name

<datatype>
Summary of the variable.

If the variable is a “switch” which can only be set using special functions (e.g., **on** and **off**) use the correct construction to set *and* reset the variable. The opening phrase of the annotation should indicate the state of the variable while the rest of the annotation indicated the action taken on that state. For instance

on **PropagateChanges**

off **PropagateChanges**

<boolean> If true, atttempt to maintain consistency throught a model (default on).

Formatting Program Command Lines

Programs are formatted like this

```
@quotation
@pindex program-name
@b{program-name} [ @i{argsList} ]
@table @i
@item Output
<datatype> Summary of program
@itemx argsList
<datatype> Meaning of argsList.
@end table
@end quotation
```

If a program accepts command line arguments (e.g., a list of files separated by commas) these are listed, if they are optional they should be enclosed in square brackets. The “datatype” of the output of a program should be specified as for functions. These are typically “RLE”, “text”, “a1 text”, “a1 binary”, or “none”.

The body of the manual is usually just typed text with few embedded commands. There are however several TEX conventions which need to be observed. Double quotes should be used in three places: when quoting a reference in the body of a paragraph, when using a term figuratively (e.g., the warp operator “pushes” a surface), and when referring to a word literally (e.g., type the word “returns”). On these occasions the double opening quotes are indicated with two backquotes “ and the closing quotes are two apostrophes ”. We also place file names in double quotes, but not shell variable references like $alib. TEX has three types of dashes: the hyphen in the middle of a word is a single minus sign “-”, the dash between two numbers (e.g., chapters 1–4, figure 8–1) is indicated by two minus signs “–”, and the dash used to break a sentence is indicated by three minus signs “—”.

Abbreviations and Capitalization

The abbreviations “i.e.” and “e.g.” are *always* followed by a comma. Sentences should always begin with an uppercase letter, but datatypes and functions should always begin with a lowercase letter. This means a sentence *cannot* begin with a datatype or function name! When writing text the auto-margin feature of your favorite editor should be used and the margin set to column 69 (not 70). This is so the on-line version won't wrap around the screen.

Here is the capitalization/spelling of some common words:

`shape_edit`	`Alpha_1`	`Nil`	`T`

```
B-spline      C-shell     C language         Rlisp
Unix          Lisp        counter-clockwise
XY-plane      X           Y                  Z
```

{Formatting Figures

To insert figures into a section use

> @iftex
> @topinsert
> @vskip 4in
> @special{psfile=*your-ps-file.ps*}
> @center @b{Figure @Yfigurenumber}: caption
> @setfigure{*some-nice-mnemonic*}
> @endinsert
> @end iftex

The "@iftex" and "@end iftex" escape into raw TEX mode. Use "@topinsert" for figures which do not occupy an entire page and "@pageinsert" for those that do (the "@endinsert" is the same either way). To get a figure number use "@Yfigurenumber". The "@setfigure" command inserts the figure number, page and chapter number into the TEX ".aux" file under the name *some-nice-mnemonic* and then increments the counter. This means the "@setfigure" should always come **after** the reference to "@Yfigurenumber". To reference this name later in the document use

> ...see figure @refx{*some-nice-mnemonic*-fig} on page @refx{*some-nice-mnemonic*-pg} which shows...

Figure numbers have the form *chapter_number-figure_number* and the figure number is reset to 1 at the start of each chapter.

3.9 Photo Guidelines

The images that we produce and publish are perhaps the most effective advertising and documentation of the capabilities of the Alpha_1 system. The composition of those images is worth some time and thought before shooting film.

Image composition

One factor which influences image composition is the aspect ratios of the various intermediate stages which an image goes through from the computer to the finished print. (In the following, all aspect ratios are stated in similar form for ease of comparison.) The path which is of main interest is from the Grinnell frame buffer (512 by 480 pixels) to the display (4 by 3) and simultaneously to the Matrix camera which has a 5 by 4 mask and makes a 4 1/2" by 3 3/8" picture on the negative (maximum). Most of these are adjustable, so there is a general lack of calibration in this process.

To make this a little clearer, we have fixed the 4 by 3 video monitor as the standard image which determines the aspect ratio of non-square pixels. If you want to draw a circle on the monitor, you must allow for the mapping of the 512x480 frame buffer onto the 4x3 display to make the circle round. (This results in a pixel aspect ratio of 1.25:1.)

Coming from the other end, printing on 11" by 8 1/2" paper there are two common formats; the Thesis format which allows ample margins for binding requires the total image area to be 9" by 6" or the "handout" format, which requires less margin, so is about 9 1/2" by 6 1/2".

There are several reasons to try for the 9 by 6 format. It can easily be expanded to the handout format during enlargement if the image is not too tight against the edges. It is also the same aspect

ratio as 35mm. slide format. Thus if the image is suitable for thesis format, it is convenient to make slides from the same negative for presentation.

The question is "What portion of that 4x3 picture on the display winds up on the print?"

A convenient answer is the 3x2 chunk out of the bottom of the 4x3 picture. On the Grinnell, this is the area covered by the bottom 427 lines. This leaves a fog-free area at the bottom of the negative for adding titles and room for a control grey scale at the top.

Composition guidelines

Fill the frame to get maximum spatial resolution out of the medium. If your picture is not square, make the long part lie along the long axis of the negative.

Center the subject matter. Compose for even distribution throughout the picture area. Don't make it too tight at the edges, however. Leave a little room for the photolab.

If you annotate the image, text or titles should fit the image. The title should not exceed the image width. The titles should not be too tight against the edge of the image area. If a series of images with titles is to be made, the image sizes should be consistent so that the titles don't have to be scaled up or down to make the pictures play well together.

Fonts with seriphs don't come through as sharply as ones without. One method for titling prints is to make captions on a hight resolution printer and make Kodalith negatives from it for crisp clean black letters.

Intensity in Images

Try to make high contrast pictures rather than dull, flat ones. If you sit in a dark room and look at a monitor, your perception of the picture is quite different from a print of the same image. Pictures are usually too dark. Use the high end of the intensity spectrum. (The one exception to this is images which are to be dithered to black and white for inclusion into documents printed on the laserwriter. These special images must have very light or white backgrounds.)

Color

No white backgrounds! It is hard to make a good, really white print. A black background or deep blue one tends to maximize the visual impact of the subject matter. Red or Yellow backgrounds bleed onto the subject. You could use light magenta or pink as a background, but a dominant background changes the way one visually perceives the subject matter.

Images which contain pairs of colors which are complementary have more visual impact.

Practice

Make a picture which fits in the right part of the screen. A program which may help with this is called "frame". Run it with a frame buffer picture and it will remove everything outside the 2 by 3 area described above. It has a set of lines surrounding the "recommended image area" which the image should fill. You can adjust the vertical position of saved images to fit using the **svfb -p** option to set their position. When you have a picture which fits the frame, put a fresh copy in the frame buffer, make the exposure with the program **wedge_expose**. It sets the portion of the picture which will not show to color 0 (normally black), exposes the picture, puts up a step wedge using a linear color map with time, date, title string, and then restores the original color map. (The picture is gone, however.) Having the step wedge on the negative helps in the photolab and is an indicator of how the hardware is working.

Shoot a color Polaroid first. Color Polaroid is about $2 per sheet, but can save making pictures of bad images. If it looks good, make a negative and have prints or slides made from it. Best way to go for many copies of the same slide set. Other slide options are to shoot Ektachrome 64 directly

and have it processed (around $10 for 36 exposures). When you want quick, low quality slides, 35mm. Polaroid works at around $15 per 36 exposures.

Prints with Multiple Images

Two negatives per page work well for thesis format. There are some restrictions related to color pictures. Both color negatives must use the same enlarger color correction (filter pack) and the same exposure time. This is a problem if you want one new and one old negative. If you want to do one color and one black and white image, the B&W must be a high contrast (Kodalith) negative. It is very hard to do a continuous tone B&W and a color image two-up.

Four images per page are tight for thesis format, but make a nice "handout" on 8.5" by 11" paper. Apply the same exposure rules as above. The negative format guidelines for 2 by 3 aspect ratio don't work well for this: the image vertical dimension must be squashed a bit. If you need to do this, write some guidelines and fix the "frame" program to help with composition.

4. Programming Tools

This chapter describes a number of utilities and programs which are generally used only by system programmers on Alpha_1. Programs which are suitable for general users are discussed in the Alpha_1 User's Manual and not in this document.

4.1 Alpha_1 Power

The following programs are to aid in getting around in places that are owned by user **alpha1**. These tools are very important, since all the important directories under "~alpha1" are not group-writeable.

chownal
: Changes the ownership of a file to be **alpha1** and sets the protections according to the standard practice. Files owned by user **alpha1** generally allow read by owner and group only, and write by owner only. Executable files allow owner, group, or anybody to execute them.

sa1
: Executes the command that follows the **sa1** as user **alpha1**. This is useful occasionally, but should be considered in the same class with super-user commands — i.e., use it sparingly and be sure you know what you're doing.

mka1dir
: Creates a new Alpha_1 directory as a subdirectory of an existing one, which must be the current directory. The name of the new subdirectory must be specified as an argument.

mknewdir
: Creates a subdirectory of the current directory. This is generally used to create working areas under an **alpha1** owned directory. The subdirectory is named by your initials if no name is specified as an argument. The subdirectory is owned by the user, rather than by user **alpha1**.

For more information about the philosophy of working in the **alpha1** owned file structure, (see section 3.3 [Working Procedures], page 12). There are also descriptions of our approach to source code management (see section 3.3.3 [Checkout Procedures], page 13) and the programs and scripts used to implement the system (see section 4.3 [Check & Friends], page 44).

(Note: Just so you know, **chownal**, **sa1**, and **mka1dir** don't really exist as such. They have become aliases in "$cmds/a1_setup.vars" which ensure that the environment variables controlling **checkin**, etc. are bound to the Alpha_1 values, and pass the arguments on to the more general **chownsys**, **subsys**, and **mksysdir**, respectively.)

4.2 New User Setup

The **setup-new-user** command makes all the other commands accessable to a new member of the Alpha_1 development or users group. It takes a single argument which is the login name of the user to install. (Warning: this can only be done successfully by a wizard with "su1" access, although it is not invoked as an **su1** command.)

It queries separately whether to do each of two phases: setting up the user's home directory and installing "dot" files. There are many system-dependent assumptions, including the assumption by the directory setup phase that the user directory should be under "/u". The directory setup includes creating a "/u" directory if none existed and copying dot files there from an old login directory if appropriate. The "/etc/passwd" and "/etc/group" files are edited if desired.

The dot file installation phase attempts to aid in merging in old user customizations by stashing old copies of the dot files in a ".old" subdirectory. Where applicable, user customizations already applied are identified by "diff"ing the old files with the "/usr/skel" files installed by the **mkusr** command when the user was first entered into Unix. Diffs are also placed in files in the ".old" subdirectory.

The new ".login" file is customized by prompting for a full name string to be placed in the USER-NAME environment variable as the author's name in standard file headers.

Finally, a message is sent to the new user's mail file summarizing what has been done and mentioning the "merge" and "Rcp" commands for completing the process of integrating old dot files and directory contents.

Setup-new-user lives in $cmds instead of $bin for two reasons. First, it really shouldn't be on the usual command search path, since it can only be run successfully by wizards and is only used rarely. Second, the "Alpha_1 standard" dot files are kept on $cmds and they should be maintained together.

The list of dot files installed by **setup-new-user** as of this writing is:

.login .cshrc .aliases .defs .a1defs .logout
.t10.fcns .emacs_pro .indent_pro .viewrc .calendar

Note: it may be necessary to delete an existing ".emacs_pro.mo" file to force a new one to be generated. Alternatively, touch the new ".emacs_pro" file.

4.3 Check & Friends

These programs arbitrate write access to source files in the Alpha_1 system. Alpha_1 group members will have read access to any file under "~alpha1" at any time, but must use **checkout** and **checkin** to obtain and release private copies of source files. Those who do not check out files before making modifications run the risk of losing the changes, as well as incurring the wrath of other members of the group!

You can invoke **checkout** and **checkin** with either files or directories as arguments. These programs will almost always be run from a subdirectory of the directory on which the files being checked out reside, although this is not required. Those unfamiliar with our methods of using working directories in this manner should read the working procedures section (see section 3.3 [Working Procedures], page 12) of the Alpha_1 standards. If you check files out while connected somewhere other than a subdirectory of their usual home directory, you will have to specify the full path name, usually using one of Alpha_1 directory variables. You will also have to specify the full path name when you check the file back in.

Please also see section 3.3.3 [Checkout Procedures], page 13 for recommended procedures in using these programs.

4.3.1 Checkout

The **checkout** command allows a user to check out a group of Alpha_1 files for modification on his own workspace. Until the files are checked back in by this user, no other user may check them out. The user may check out either individual files, or the entire contents of any particular directory. It is strongly recommended that all the files checked out at once be part of the same parent directory, because your working copies will **all** end up on the currently connected directory. (**Checkout** can

handle grabbing them from different places just fine — the problem is that **you** have to remember where they came from in order to check them back in).

checkout [**-m**] { *file*... | **-d** *dir* }

Output	<none> Checks files out to this user, copying them to current directory.
-m	If specified, uses the same checkout reason for each file.
file	The files to be checked out.
-d *dir*	If specified, an entire directory may be checked out at once.

If any of the files specified in the argument list to **checkout** do not have a full path name (as will usually be the case), then the file is expected to reside on the parent directory of the currently connected directory, and the full path name is completed appropriately. All file paths are resolved relative to the root of the Alpha_1 file tree, "~alpha1", and only the portion of the path below that point will be used within the source librarian.

The **checkout** command checks that the files being requested are not currently checked out by any user. If none of the requested files have been previously checked out, and if all of the requested files exist, the request is granted, and the files are checked out to the current user. The **checkout** process leaves the user with a copy of the file(s) on the current directory which he is allowed to modify.

When the **-d** option is given to check out a directory, **checkout** assumes that the user wishes to check out the entire contents of the directory. This currently means all text files which reside on the named directory (except ".%*", ".#*", "*.bak", "*.old", "*.err", and "tmp.*" files). Subdirectories are **not** included (i.e., **checkout** is not recursive). If the directory contains **no** such files **checkout** will tell you so, but will still create a database entry indicating that you have the directory checked out.

The **checkout** program requests a reason for the checkout of each named file or directory, and stores this in a simple database. This reason will be included in the history record which is created when the file or directory is checked back in and may be changed with **checkedit**. The **-m** argument causes multiple files to be checked out with a single reason, i.e., the reason is read from the terminal only once.

Note: The **checkout** procedures are equipped with a lock file so that two users do not simultaneously attempt to modify the checkout database. If the database is in use when a request is made, the program waits for the database to be freed. There is an automatic mechanism for unlocking the lock file if **checkout** aborted, which should be sufficient on a Unix which provides file locking primitives like Berkeley 4.2bsd or Apollo AUX. On other Unix versions, it may be necessary to delete the file "/lock/alpha1" to free up the database if a system crash occurred while the lock was set. This should be done only when you are absolutely positive that it is really locked erroneously, and that no one else is accessing the checkout database.

4.3.2 Checkin

The **checkin** program checks files back into the system. **Checkin** takes a list of destination files or a directory as its arguments. Since the new versions must be copied back into the system, all of the named files (with any path names stripped) must exist on the currently connected directory, and be checked out if they existed previously. The current directory must always be a working directory and the destination directory must always be owned by **alpha1**. If the conditions on **checkin** arguments do not hold, nothing is done; a checkin command will never partially do its work.

checkin { *file...* | **-d** *dir* }

Output	<none> Checks in the named files, installing them in the source tree and recording the changes which were made.
file	The files to be checked in.
-d *dir*	If specified, an entire directory can be checked in at once.

If the named files exist on the current directory, but not on the system directory, then they are assumed to have been just created, and will be installed along with any modified files.

As in **checkout**, the **-d** option may be given to check in a directory, in which case all the text files (see the description of what a text file is in the previous section) on the current directory are installed on the named directory. Be sure that extraneous files aren't lying around when running **checkin** on directories so that we don't end up with a lot of garbage "installed" in the system. A directory to be checked in must have been checked out.

Checkin, after approving the request subject to the conditions described above, retrieves the database entry which describes why the file was checked out. This text, together with some other information such as the user's name and the checkout and checkin times, is placed in a file whose name is a unique transaction number on a special Alpha_1 directory. The revised file is also compared against the original (with **diff**) and the changes made are recorded in this transaction file. For files which are being checked into the system for the first time, a creation notice is put in the transaction file.

Each file is then copied back to the master directory. The working directory copy of the file is renamed with a prefix of ".#", like emacs backup files, and will go away automatically after a while. There is currently no provision for storing backup copies of the source files — the **diff** listings provide the mechanism for returning to a previous state if necessary.

The **checkin** program uses the same locking mechanism as **checkout**, since both programs share the same database.

Currently, there is not an automatic and flexible way of accessing the transaction records, but the implementation has left room for easy addition of various methods of accessing the data, including periodic archiving. Currently, an index is maintained which users may examine to identify the transaction number of an entry in which they are interested. The **checkhist** command (see section 4.3.7 [Checkhist], page 48) may also be used.

A "global make" facility (**makeall**) is used to propagate changes from source files into Alpha_1 libraries and programs. It may be run automatically from **cron**, or manually by an Alpha_1 system maintainer. Thus, it is not necessary to run **make** on the Alpha_1 directory which your files have been checked into. It suffices to add new files to the makefile and let **makeall** do the rest.

At the time a file is checked in or retired, a distribution message is automatically sent to other Alpha_1 installations to update their source directories. For more information, See section 4.3.9 [Distribute], page 49.

4.3.3 Check

The **check** program gives you information about files that are checked out. A simple list of file names (with complete paths) and users who have checked them out is maintained by the **checkout** and **checkin** programs. The **check** program merely searches this list for the occurrence of particular strings.

check [**-y**] { *file* | *user* }

Output	<text> A listing of source files which are currently checked out.
-y	If specified, the checkout reasons which were given for each file will also be printed.
file	If given, see if these files are checked out.
user	If given, see what files this user has checked out.

Check can be used to find out:

- what files any particular user has checked out, as in **`check cobb`**
- who has checked out a particular file, as in **`check R_poly.c`**
- what files on a particular directory are checked out, as in **`check $rendd`**

If given a **-y** flag, **check** will tell you why each file was checked out by printing the checkout database entry for it.

4.3.4 Retire

The **retire** program is sometimes useful. **Retire** is used to get rid of old source code. The files to be retired may be checked out. Instead of checking them back in, **retire** should be run on those files. **Retire** can also be run on files which are not currently checked out.

retire *file...*

Output	<none> Remove an installed source file.
file...	The files which are no longer needed.

Retire is quite similar to **checkin**, except that it inserts an obituary notice in the transaction file, and puts a copy of the corpse in the diffs directory. The source files on the master directory and the currently connected working directory are both deleted.

At the time a file is retired, a distribution message is automatically sent to other Alpha_1 installations to update their source directories. See section 4.3.9 [Distribute], page 49 for more information.

4.3.5 Checkedit

Occasionally, the reason you gave for checking out the file will need to be revised when the modifications are actually completed, or while they are in progress. Often, one discovers that there is more that needs to be done than was initially realized, or wishes to add more detail about what was actually changed and why. The **checkedit** program can be used to edit the reason database.

checkedit [**-s**] [**-S**] [**-m**] { *file...* | *dir* }

Output	<none> Retrieves the checkout reason for a file so user can edit it.
-s	If specified, just get new reason from standard input.
-S	Like -S, but print the old one first.
-m	Use a single new reason for all the files.
file...	The files for which checkout reasons are to be edited.
dir	A directory for which checkout reasons are to be edited.

The program takes the same types of arguments as **checkout** and **checkin**, filling in path names for files if necessary. All of the files or directories must actually be checked out by you before you can edit the checkout reason.

The latest entry for each file or directory named in the argument list is extracted from the database and you are placed in an editor of your choice with that text. When done editing, save the file back out, and the database will be updated by **checkedit**. The revised reason will be placed with the change history of the file when it is checked back in.

The **-s** argument gets the modified reason from stdin and just puts it into the database. This is useful in emacs C-shell windows, where you don't want to run another editor underneath and may still have the original reason in a dialog file with the original **checkout** command. The **-S** option is like **-s** but also prints the old reason before reading in the updated reason. The **-m** option is as in **checkout**, causing a single new reason to be applied to all of the files.

4.3.6 Uncheckout

Uncheckout is used to reverse the effects of a **checkout**. The files are marked as being no longer checked out, and the copy on the currently connected subdirectory is deleted. This is for use only if **no changes** are being made to the file. If you made changes which need to be installed, you should use the **checkin** command. **Uncheckout** says, "I didn't really mean to check it out" and keeps no records of the fact that the file was temporarily checked out.

uncheckout { *file...* | *dir...* }

Output	<none> Release the checkout lock on a file without making any changes.
file...	The files to be released.
dir...	A directory to be released.

4.3.7 Checkhist

Checkhist is a mechanism for finding out the history of files in the Alpha_1 system. Obviously, the history only goes back as far as the installation of the source librarian system (June 1982).

checkhist [**-fn**] [*-count*] *file...*

Output	<text> A listing of the changes which have been made to a file.
-f	If specified, full **diff** listings are printed.
-n	If specified, print the name of the file containing the **diff** listing.
-count	Only go back this many entries.
file...	The files to check history of.

The **checkhist** command prints the selected transaction file entries, in reverse chronological order. This includes information such as who checked the file out and when. If run with the **-f** flag (full history), **checkhist** prints the **diff** file which was produced for each transaction. These files record the reason the file was checked out, and the differences between the new version and the previous version. If run with the **-n** flag instead of **-f**, **checkhist** prints the name of the diff file which was produced for each transaction. This is useful for more serious examination of the files, and especially facilitates using an editor to compare them.

The *-count* option (a dash followed by a decimal number) may be given to control the maximum number of entries to list. The default is 10.

For occasions where you want to know about more than one file, it may be useful to know that the file arguments are actually strings that are matched against the hisory index entries to select the set that is displayed. Each entry contains the filename, user name, and date of the checkin. Only arguments that start with a slash must actually resemble real filenames under "~alpha1".

4.3.8 Checkoutdb

Checkoutdb allows a user to checkout an emacs database in order to make modifications without colliding with other users. The only current use is to checkout the Alpha_1 info database for updating the documents. **Checkoutdb** gives the user write access to the various files in the database, but these files must be modified only in carefully controlled ways. **Never** just edit the text files! Once the user is finished modifying the database, the files can be restored to normal Alpha_1 access with the **chownal** command.

checkoutdb *dbname*

Output	<none> Checkout an emacs database by changing the owner of the files.
dbname	The name of the database file (without the ".dat").

4.3.9 Distribute

At the time a file is checked in or retired, a distribution message is automatically sent to other Alpha_1 installations to update their source directories. **Checkin** and **retire** prepare a file of C-shell commands which can be run on a remote machine to duplicate the changes to the file structure on the local machine. Then they invoke the **distribute** script to transmit the commands.

distribute *subject* < *change-script*.csh

Output	<none> Distribute a change to remote sites.
subject	Description of the change.
change-script.csh	Commands which effect the changes.

Currently, the distribution is done via mail messages. The change command file, which is read on stdin by **distribute**, is bracketed with known strings so it can be stripped out of the body of the message automatically. The subject of the message is the first argument to **distribute**, typically a filename, prepended with a counter to help insure that the order of changes is preserved.

Change messages go to mailing list **a1_distribute** from the master site and go to mailing_list **a1_chg_req** from a remote site. The intent is that all changes and additions to Alpha_1 from remote sites will be filtered through the master site, where they will be checked in and thus distributed to all the other sites. We would like to encourage bug fixes and enhancements from remote sites, and will try to give appropriately fast turnaround to change requests.

A user should invoke the **distribute** script manually only when the usual tools of **checkout**, **checkin**, and **retire** are inadequate to perform the necessary changes. By definition, this occurs in cases which do not (have not) occurred much, or the tools would already have been built. For instance, there is currently no mechanism for renaming files en masse, and it is a bit clumsy to check them into the new location and retire them from the old. This could be done with **sal** and duplicated remotely with **distribute**, or a rename script could easily be written to do the work instead of doing it by hand each time. Users should also be very certain they know what they are doing when they run **distribute**.

Distribute Template

Once you have decided to use **distribute** manually, there is a template that eases using it in file "$bin/lib/distribute.template". It is designed to be pasted into an emacs C-shell input buffer (by convention saved as ".cshin" files). In order to assure that your commands work properly before distributing them, the first commands are

```
sal csh
set prompt = '! alpha1# '
alias z suspend
```

which start up a C-shell running as user **alpha1** under the normal one in your C-shell dialog window, and set the prompt string to remind you which shell you are in.

The "z" command is a short name for the "suspend" command that pauses the **alpha1** C-shell and goes back to your normal one. You can continue a suspended subshell with the C-shell "%" command (normally aliased to "c" at Utah.) In particular, you should do **checkout** and **checkin** actions as yourself, not as user **alpha1**, since the **checkin** will distribute its own change messages.

You should fill in a description of the changes in the lines:

```
=distribute "...Brief reason" << 'EOF'
# ...Full description of reason for changes.
```

Replace the "...Brief reason" with only a few words, since that string will be part of the subject line when the change message is distributed. Put in C-shell comment lines starting with "#" to fully describe what you are doing to help the Alpha_1 maintainers who will be receiving the commands and executing them via **install-a1-chg**.

Then, after some setup commands, comes the line:

```
...csh commands to accomplish the change.
```

which you replace with the actual commands you will want executed on all of the Alpha_1 sites. These are the commands which you should first execute in the **sa1** C-shell to test them for correctness, modifying the copy of Alpha_1 at your site at the same time. Moreover, you should keep the command script to be distributed in order so that it can be automatically executed on the remote sites, without intervention.

You should use Alpha_1 "$variable" references to directories rather than absolute file paths in your commands, since Alpha_1 may be installed in a different place in the file system of other machines. You should also comment the command script to explain the actions.

Sometimes, updates involve sending new copies of files which are not under Alpha_1 **checkout** and **checkin** control, like pieces of PSL. You can insert copies of the "$bin/lib/distribute-file.template" file, which brackets each file you want to send just like **checkin** does:

```
cat << 'distribEOF' > ...where
...what
'distribEOF'
```

You should use a **cp** command or something on the local host, put the **cp** destination in place of the "...where", and paste the file in place of the "...what" placeholder. Consider wrapping chunks of the script in conditional logic, e.g., to gracefully account for sites that may not have PSL installed.

Once you have a good command script, which has been executed a bit at a time on the local machine, you are ready to distribute it. Go back to the "=distribute" command line and delete the "=". (It was left there as a failsafe to avoid distributing the commands prematurely.) Make sure you are in your own shell, rather than the Alpha_1 shell. Then execute the commands from the

```
distribute "What is being done" << 'EOF'
```

line to the

```
'EOF'
```

line, and your changes will be sent off to all of the remote Alpha_1 sites to be executed there.

4.3.10 Install-a1-chg

The **install-a1-chg** script is intended to handle the common part of installing a change message from the **distribute** script on a remote site. **Install-a1-chg** is a filter which takes a change message on stdin, strips the change script out of the message body and runs it through an **sal** C-shell while echoing the commands, and mails the results to mailing list **a1_dist_ack** and returns them on stdout.

install-a1-chg

Output	<none> Execute an Alpha_1 change message.

There are currently gemacs packages available for the rmail and mh mail reading modes which bind the "I" key to install messages. **Install-a1-chg** can be invoked from any mail reader which can pass messages into a Unix command.

4.4 Makefile Tools

This section describes several of the tools which are used in generating makefiles from "makefile.spec" files and creating dependencies in the makefiles which insure that recompilation is done when needed. If you are not familiar with "makefile.spec" files, read the section on make in the Standards chapter first (see section 3.6 [Make], page 22).

4.4.1 Gen-makefile

The **gen-makefile** program configures a makefile by running a makefile source file through the **preprocess** program under control of the "a1_config.h" and "dev_config.h" files. It then expands the results with **expand-makefile** to turn abbreviated symbols into complete makefile lines.

gen-makefile *-h -w -d -t -s -b X*

Output	<none> A new makefile is created.
-h	Force home directory expansion.
-w	Force working directory expansion.
-d	Force debug directory expansion.
-t	Preserves the "/tmp" workfiles for examination in case of problems.
-s	Puts the new makefile on stdout instead of into file "makefile".
-b X	Sets the C binary extension to *.X* (e.g., "-b b" for Mac makefiles.)

The "-h" flag is also used by **config** on $utild, since the **owner** command may not be built yet.

The makefile produced differs depending on whether it is expanded on the home directory, working directory, or a debug directory.

- Home directories are owned by the same user as the Alpha_1 root.
- Working directories are typically made by the **mknewdir** command under Alpha_1 home directories and are owned by the programmer.
- Debug directories are subdirectories of Alpha_1 library or program directories and are named "debug". The C code is compiled with debugging information on debug directories. (Debug compilation precludes optimized C compilation, which is normally done on home directories.)

Preprocessor commands are introduced by percent (%) signs instead of pound (#) signs since pound signs denote makefile comment lines. Preprocessor symbols **HOME_DIR**, **WORK_DIR**, or **DEBUG_DIR** are set to 1 when **gen-makefile** is run on home, working, or debugging directories respectively, and are 0 otherwise.

The preprocessor symbols are intended to control sequences like:

```
%if HOME_DIR
    # Standard compilation includes optimization.
    CFLAGS = -O $(IFLAGS)
    LIB = A1_LIB
%else
    HOME = ../
    # Workdir and debug compilation and loading include debugger info.
    CFLAGS = DBG_FLAGS $(IFLAGS)
    LIB = A1_LIB/debug
%endif
```

In addition to standard makefile lines, the "makefile.spec" files can contain abbreviations that are expanded into a full makefile by the **expand-makefile** program. Configuration preprocessing (substituting in the "a1_config.h" and "dev_config.h" definitions) is done before makefile expansion.

"Makefile.src" files are just normal makefiles with preprocessor commands, but **expand-makefile** is not run. "Makefile.spec" files should be used in all Alpha_1 directories except $utild, the directory containing the **expand-makefile** program. The $utild directory is compiled first in the process of building an Alpha_1 installation.

An optional argument to **gen-makefile** may give a file path to the source file, otherwise default locations searched in order are:

- local "makefile.spec" or "makefile.src" file on the current directory,
- "makefile.spec" or "makefile.src" on the parent directory.

4.4.2 Preprocess

Preprocess is a little script intended to increase the portability of Alpha_1 by encapsulating the invocation of the C preprocessor used by **gen-makefile** and maybe other places. Arguments are passed through to the C preprocessor, for example **-I** arguments for include directories to be searched for ".h" files.

The standard version of **preprocess** uses "CC -E" on a temporary file, which is the most likely method to work on various Unix systems. (It is probably a botch to use the C preprocessor instead of **m4** for the makefile expansion job, but it works pretty well for now.)

4.4.3 Expand-makefile

Expand-makefile is a program used by **gen-makefile** to expand Alpha_1 command templates in a "makefile.spec" file. Some fairly complicated processing is done to produce nice, intuitive results from simply specifying groups of C file names to compile and groups of compiled files to link into programs. **Expand-makefile** reads the makefile source from the standard input and puts the expanded results on the standard output.

expand-makefile [**-wd**] [**-l**] [**-o** *o_home*] [**-b** *X*]

Output	<none> Expand command templates in a "makefile.spec" file.
-w	(work_dir) Indicates makefile is being expanded on a work directory.
-d	(debug_dir) Indicates expanion for a debug directory. If neither **-w** or **-d** is given, the expansion is made for a home directory.
-l	Indicates a library directory; otherwise expand for a program directory. This flag will also be set if there is a **LIB_DIR** line in the source.
-b *X*	Sets the C binary extension to .*X* (e.g., "**-b** b" for Mac makefiles).

The input stream contains normal makefile lines with "template commands" intermixed. Makefile lines are:

- comment text lines (blank or start with a "#"),
- variable assignments (start with a word followed by an "="),
- entries (line with one or more target files, a ":" or "::", and a prerequisite file list, followed by an optional set of action lines starting with tabs.)

Variable assignments and target lines may have trailing comments but action lines may not, including action lines joined to target lines by a semicolon (";"). Makefile lines may start with blanks and are continued by a backslash at the end of the line. Makefile lines are "prettyprinted" as a side effect of **expand-makefile**.

Template commands have a keyword at the beginning of a line and have fields separated by whitespace (one or more blanks, tabs, or newlines.) Backslashes (\) at the ends of lines to be continued are not necessary in template commands, unlike normal makefile syntax. The template command words continue until a line starts with another template keyword or is a makefile line.

Makefile Template Command Summary

LIB_DIR sets the lib_dir flag, which causes the **CFILES** list to be expanded appropriately for library directories.

LIB_O_DIR sets the lib_o_dir flag, which is intended for working directories of libraries where ".o" files are being built separately rather than combined into a ".a" library file. These directories are like program working directories, except that the **OBJS** lists are culled based on whether the sources are on the directory.

The **CFILES** template specifies a named C source file list, from which **CFILES** and **OBJS** lists are generated.

On a home directory, the **CFILES** and **OBJS** lists contain all of the specified files, with ".c", ".y", or ".l" suffixes converted to ".o" in the **OBJS** list.

On a working directory, only source files which are present or will be generated on the working directory are put in the **CFILES** list.

On a library working directory where you want just the ".o" files to override the library, the **OBJS** list contains ".o" files corresponding to the culled **CFILES** list. On a program (non-library) working directory where you want to load the full set of object files into a program, the **OBJS** list is the full list with "$(OHOME)" prefixed to the ".o" filenames which are not present on the working directory.

On a debug directory, "$(HOME)" is prefixed to the ".c" files and cc entries are generated, since the ".c.o" **make** rule won't work if the target file is on a different directory from the prerequisite file.

For example, consider:

```
CFILES  name    pgm.c util.c
```

On a home directory this, would expand into:

```
nameCFILES = pgm.c util.c
nameOBJS = pgm.o util.o
```

On a library working directory (assuming "pgm.c" was present and "util.c" wasn't), it would become:

```
nameCFILES = pgm.c
nameOBJS = pgm.o
```

On a program work directory, it becomes:

```
nameCFILES = pgm.c
nameOBJS = pgm.o $(OHOME)util.o
```

On a debug directory, we would get:

```
nameCFILES = $(HOME)pgm.c $(HOME)util.c
nameOBJS = pgm.o util.o
pgm.o: $(HOME)pgm.c
        $(CC) $(CFLAGS) -c $(HOME)pgm.c
util.o: $(HOME)util.c
        $(CC) $(CFLAGS) -c $(HOME)util.c
```

References to "$(nameCFILES)" and "$(nameOBJS)" will be accumulated and automatically appended to the makefile "CFILES =" and "OBJS =" lists. Lines of the form

```
$(FILECMD) name $(nameCFILES)
```

will be inserted after a "files:" line (mainly for **alcount**.)

LFILES is like **CFILES**, specifying (R)Lisp source files to be compiled. Currently, only ".r" suffixes are supported for compilation into ".b" files. Other suffixes, (e.g., ".lap") are assumed to be interpreted files. **LFILES** source file lists are generated, as well as **BINS** lists with the ".r" suffixes changed to ".b".

On home directories, **LOADS** lists with "$(LOAD)/" prefixed to the ".b" or ".lap" name are generated, as well as an installation entry for ".lap" files that copies them to **$(LOAD)**.

References to "$(nameLFILES)", "$(nameBINS)", and "$(nameLOADS)" will be accumulated and appended to the makefile **LFILES**, **BINS**, and **LOADS** lists if they exist.

The lisp source files are added to the "files:" section as well:

```
$(FILECMD) name -lisp $(nameLFILES)
```

PGM specifies loading of a C main program.

```
PGM  name       $(nameOBJS) $(LIBES)
```

produces:

```
name: $(nameOBJS) $(LIBES)
        -cp name name.old
        $(CC) $(CFLAGS) $(nameOBJS) $(LIBES) -o name.new
        mv name.new name
```

OPTABLE specifies generation of a generic operation dispatch table. A customized dispatch table is needed by a program which makes generic operation ("_obj") calls other than the basic operations for the StrgMgmt, BinIO, and Debug protocols. The optable is generated from an object

management "-specs" file giving the objects and operations to include when there are no "local objects" or "local protocols" defined.

```
OPTABLE name
```

Optional local objects and/or local protocols may be specified by:

```
OPTABLE name -O oname -P pname
```

The "oname" and "pname" default to "name", and either of the options may be omitted.

LOCPTCLS and **LOCOBJS** specify object management entries for local protocols and local objects. For local objects, a local "-str.mstr" file containing a set of ".str" files to process is converted to a local "objs.mstr" file and a local "-objs.h" file. For local protocols, a local "-ptcl.mstr" file is converted into ".h" and ".c" files defining generic dispatching routines for the locally defined protocols.

```
LOCPTCLS pname
LOCOBJS oname
```

HOME Variable Processing

All filenames prefixed by a question mark are treated as files which may be referenced from the home directory or debug directory while working on a subdirectory. On home directories, the "?"s are simply ignored, and on debug directories they are always translated to a "$(HOME)" prefix.

On working directories, "?file" is changed to "file" if the file is a regular file on the working directory, or if there is an entry in the makefile which will generate the file from other files which **are** present on the working directory. Otherwise the "?file" is changed to "$(HOME)file" (or "$(OHOME)file" in the case of ".o" files.) This applies equally to file lists and makefile entries generated from "template commands".

4.4.4 Make-depend

Make-depend is a command to automatically generate dependency entries for C programs in makefiles. It determines in which directory the ".h" files from "#include" statements will be found and inserts that dependency information in a section at the end of the makefile. It also generates makefile actions which cannot be expressed as default **make** rules.

The normal way of running **make-depend** is based on the fact that the standard Alpha_1 makefile templates include an entry for doing this, which passes in an appropriate list of files and include directories to be searched. So you should only have to say

```
make depend
```

and the makefile will update its dependency section using **make-depend**.

Make-depend allows makefiles to be used on working subdirectories of the controlled Alpha_1 source directories when parts of the sources are being developed or changed and the new code tested. You should run it on the working subdirectory makefile whenever you generate a makefile there, when you construct a new include subdirectory referenced in your working makefile, and whenever you check out a source file to a working subdirectory or add a "#include" statement to a C file.

Make-depend is also run automatically by **makeall** whenever the makefile in a source directory is updated, insuring that the Alpha_1 makefiles and thus the libraries and executables are synchronized with source updates.

How Make-depend Works

Make-depend will make a dependency line in the makefile for each C program file which includes files from one or more of the Alpha_1 include directories. The dependency information will be placed at the end of the makefile, following a line which says

```
# DO NOT DELETE THIS LINE
```

If such a line does not already exist, it will be appended to the end. If the makefile already has a line like this, everything following it will be deleted before the new dependency lines are put in (i.e., the old dependencies will be replaced). Any special dependencies a file has should be listed on separate dependency lines preceding the "DO NOT DELETE" line. (There is no problem with declaring dependencies for a file in multiple places in a makefile. However, the **make** program will give an error if there are multiple actions sections for a single file.)

Make-depend generates dependency lines from the "#include" lines in source files. They inform **make** that when the definitions in a header (".h") file change, the ".c" files that "#include" the ".h" file should be recompiled. The ".o" files "depend upon" the ".h" files, and the ".o" files may reside in a ".a" library.

Actually, there is a little more to it than that. If the ".o" files depended directly upon the ".h" files, then **any** change to a ".h" file would force a lot of recompilation since the time on the ".h" file would be newer than the times on the ".o" files. Since Alpha_1 is a large system, we choose to allow "transparent changes" which do not force system-wide recompilation. To do this, a zero-size file ".foo.t" is paired with each system ".h" file "foo.h". Conceptually, the ".t" file is a "time tag" for the ".h" file. If a ".t" file is present, it is used instead of the ".h" file in the dependencies generated by **make-depend**.

Because of the ".t" mechanism, we can encourage transparent changes to ".h" files. Changes in comments or formatting are obviously transparent. Perhaps not so obviously, extensions to a package represented by a ".h" file are transparent to the older parts of the system where the ".c" files do not use the new declarations in the ".h" file. Anyone who makes a **non-transparent** change to a ".h" file is responsible for executing **touchal** on the corresponding ".t" file when the changed version is checked in, thus forcing recompilation and relinking of functions and programs which are affected.

make-depend [**-h**] [**-m** *makefile*] [**-I***directory* ...] [**-l***lib*] *files*...

Output	<none> Add dependency entries to the makefile.
-h	Indicates that the program is being run in an include directory.
-m *makefile*	Name of a makefile to use instead of "makefile".
-I*directory* ...	List of include directories to search.
-l*lib*	Indicates that the program is being run on a library directory.
files...	The files for which dependency information should be generated.

To specify a different set of directories to be searched for the ".h" files, one or more **-I** flags may be given. These work identically to the **-I** flags for the C compiler and are by convention put in the "$(IFLAGS)" makefile variable. If no **-I** arguments are specified, the default specification consists of the "general purpose" Alpha_1 include directories under $ai.

For library directories, the optional **-l** flag is followed by the name of a library. The syntax is identical to the Unix loader flag which you specify during the load phase of a cc command, so "-lA1" specifies library (archive) file "libA1.a". **Make-depend** generates actions containing a cc statement to do the compile and then a "touch didwork" statement. "Didwork" eventually causes "updatelib" to bundle all of the new ".o" files into the library file and **ranlib** it to update the library

index.

Normally, **make-depend** searches for dependencies in the include statements of ".c" files and produces ".o" file actions. The **-h flag** is used when applying **make-depend** to an include directory where there are dependencies between ".h" files. This occurs when packages are "layered around" other packages, as in the set d_spline.h, refine.h, mesh.h, and matrix.h, each of which uses the next lower level by including its ".h" file.) In an include directory, the targets are ".t" files, which are touched by makefile actions when ".t" files for packages on which they depend are touched.

The target "all_tfiles:" is also generated, which depends on all the ".t" files in the directory. This passes on the information gained by calling "make-depend -h *.h" in the "depend:" entry, and avoids manual maintenance of a list of ".h" files in the makefile. The default make action on include directories is "all_tfiles:" so the global make does this maintenance automatically.

4.4.5 Rlisp-make-depend

Rlisp-make-depend, for directories containing Rlisp ".r" files, is the counterpart of the C **make-depend** program. **Make-depend** scans for lines containing "#include", and **rlisp-make-depend** scans for "bothTimes load" statements.

rlisp-make-depend [**-t**] [**-m** *makefile*] [**-L** *loadDir*]... *files*...

Output <none> Generate dependency lines for a makefile on an Rlisp source directory.

-t If specified, indicates that the program is being run on the include directory.

-m *makefile*
The name of a makefile to use instead of "makefile".

-L *LoadDir*
The list of directories to be search for load files.

files The files for which dependency information is to be generated.

The **-t** flag is similar to the **-h** flag of **make-depend**. It is used on the $shi directory containing all the files which are loaded into other **shape_edit** files at compile time (via "bothTimes load" statements). If present, it causes ".t" file touching actions with appropriate dependencies to be constructed for all ".r" files with "bothTimes load" statements, as well as an "all_tfiles:" target that activates them.

The **-L** flag is similar to the **-I** flag of **make-depend**. It is used to specify additional directories to be searched for the loaded ".b" or ".lap" files, or their ".t" time tag files. The current directory is always searched first by the PSL load command, then the PSL system directories, and finally the directories specified by the **-L** flag.

Note that since **rlisp-make-depend** is a simple-minded C-shell script using "grep", you **may not** separate a load statement onto multiple lines:

```
bothTimes load package1!-df,     % Don't do this.
               package2;
```

Rlisp-make-depend generates compilation actions for all ".r" files which install the completed ".b" files on the $load library directory. This is similar to the compilation actions generated by **make-depend** on C library directories.

There is some conditional logic in "$ai/lisp-defs.h" which sets the "$(DEST)", "$(BUILDFLAGS)" and "$(DESTFLAG)" makefile variables so that ".b" files are installed from home directories into $load but left in place on working directories.

4.5 Modeling Utilities

Many of the programs described in this section are fast becoming obsolete. The functions are available within **shape_edit** and should be used there. These programs remain for historical reasons, for occasional use, and because they are sometimes needed for converting Alpha_1 data to other formats.

4.5.1 Tran

The **tran** program is used for flattening a tree of transformations and objects. The program maintains an internal transformation stack, as specified by **view_trans, push**, and **pop** objects in the input stream. **Tran** echoes each object as it is read in, first multiplying it by the current object transformation. Screen transformations are ignored.

Use the "tran" keyword to load the commands with **getcmds**, as in "getcmds tran".

tran *files*

Output	<binaryFile> A transformed version of the input data.
files	<textFiles> The geometry to be transformed.

4.5.2 Flipmesh

The **flip_mesh** program reverses the orientation of all surfaces in the input, and echoes them (and any other objects in the file) back to the standard output. The orientation is reversed by reversing the order of the rows or columns of the surface control mesh. It is highly recommended that surfaces be correctly oriented using **reverseObj** when created in **shape_edit**, rather than using this program. For data not created in **shape_edit**, this may be the only option.

This program is occasionally used to process input for the intersector, where orientation matters. It is not generally necessary for the **render** program, because the **flip** flag to **render** accomplishes the same effect in shading, assuming that all of the surfaces to be rendered have consistent normal directions. (A back_color attribute can be added to make the surface normal directions visible during rendering.

Use the "flip" keyword to **getcmds** to load the commands for **flip_mesh**.

flip_mesh *files*

Output	<binaryFile> Reverse orientation of all surfaces in the input.
files	<textFiles> The data to be reversed.

By default, the rows of the surface are reversed. This can be changed by using the **cols** and **rows** switches.

cols	Set columns to be the default direction reversed.
rows	Set the default back to rows.

Implementation Details

The **flip_mesh** program is really just an I/O wrapping for the **flip_surface** subroutine in libA1. **Flip_surface** reverses several other attributes of a surface object as well as reversing the control mesh.

The knot vector in the direction being flipped is reversed. The returned knot values have the same parametric range and are restored to ascending order. If present, the contents of **surf_corner** and **edge_flag** attributes used by the subdivider are reversed, **LEFT** to **RIGHT** if the direction is **ROW**, and **TOP** to **BOTTOM** if the direction is **COLUMN**.

If present, adjacency attributes are flipped. If rows are being flipped, the **LEFT** and **RIGHT** adjacency string attributes are interchanged. If columns, **TOP** and **BOTTOM** are interchanged. Furthermore, in any of the **TOP**, **BOTTOM**, **LEFT**, or **RIGHT** attributes present, a new attribute string is constructed with edge codes transformed to reflect a similar flip on the named surface. This assumes that all surfaces mentioned are being flipped as well as the one currently in **flip_surface**. An entire object must be flipped, where an object is a set of surfaces with adjacency relationships joining the edges.

4.5.3 Isolines

The **iso_lines** program generates iso-parametric lines for polygons (derived from surfaces) which have been tagged with uv-coordinate values at their vertices. This kind of data would typically come from the combiner.

Lines can be generated in either the u or v (or both) parametric directions, and the spacing between the lines may be specified. The spacing defaults to 0.1 (in parameter space) which is probably too large a value in general.

The commands are loaded with the "iso" keyword of **getcmds**.

iso *files*

Output	<binaryFile> Genearate iso-parametric lines for polygon data.
files	<textFiles> The polygon data which has uv-coordinate tags.

Switches which may be set are:

uiso	Generate lines in the u direction only.
viso	Same for v direction.
bothiso	Generate lines in both parametric directions.
uincr val	Set spacing between the u direction lines to value.
vincr val	Same for v direction.
isostate	Display the values of switches which are different from the defaults.

4.5.4 LineMap

The **linemap** program is very similar to the **iso_lines** program. **Linemap** processes a set of polygons from a surface with uv-parameters and data values associated with each of their vertices to produce a set of evenly spaced lines of constant data. The output of this program can be used as input to **hidden**.

The commands are loaded with the "mapper" keyword of **getcmds**.

linemap *files*

Output	<binaryFile> The original polygons plus isoparametric lines for use in the **hidden** program.
files	<textFiles> The polygon data which is to have lines added.

Program switches for **linemap** are:

dspacing	Use default spacing (10.0).
spacing s	Set the spacing between data lines.
nomapdbg	No debugging information is printed (default).
mapdbg	Turns on debugging.

To see the status of the program options use

mapperstate

5. Object Management in C

This chapter contains information about the object management implementation in C for Alpha_1. The introduction is essential reading for those who have to understand how Alpha_1 C code handles objects. The application programs section is vital for those who need to write programs that work with objects. The final section describes implementation details and philosophy which may not be needed by many programmers, although the file formats will sometimes be necessary for reference.

5.1 Introduction to Objects and Protocols

The Alpha_1 system handles objects in a way that roughly mimics other objects packages like the one used in **shape_edit**. and an expanded set of semantic actions in a regular way. The object management system eliminates a great deal of "dumb" code by adding some information to the common object data structure and by letting the system "know" something about object structure definitions. The system generates some simple operations which depend only on the object data structure element types.

An *object type* corresponds to an object type in **shape_edit**, and basically consists of a data structure definition for the state of the object. An *instance* of the object type is simply an actual data structure with the state fill in for the particular object. There is a notion of *generic operations* or *methods* in the C objects package, although those terms weren't used in its development. A generic operation is one that can be applied to a number of objects. The operation is not specific to the object type it is being applied to, even though the details of the implementation may be unique for each object type. For example, the meaning of "copy" or "free" does not change depending on the type of object to which it is being applied. Usually the calling sequences of such operations are fixed.

In the current Alpha_1 C programming environment (unlike in Lisp), there is a distinction between objects and pointers to objects, and the primitive types of the C language are statically typed at compile time by declarations instead of being tagged at run time. Since we are providing a higher level of tagged structures, the term *object* will mean exclusively the tagged abstract object types. The word *type* will also include lower-level or primitive types which may be used as elements of the object data structure, where the type is known during compilation. Examples of primitive types are floats, ints, and typedef'ed types. Lower-level types at this time are untagged C structures, dynamically allocated and linked into data structures by base pointers. Examples include B-spline curves and surfaces, polygon vertices, control meshes, knot_vectors, and matrices.

A *protocol* contains a group of generic operations such as storage management, object I/O, and rendering and line-drawing display operations. Objects may *subscribe* to a protocol, meaning that the required operations are provided Objects may subscribe to a set of independent protocols The addition of protocols to the system and to individual object types is incremental.

The rest of this section describes at a high level the Alpha_1 mechanisms for implementing in C the concepts described above.

Objects in C are defined in a ".str" file which has a special format similar to C structure definitions. From the ".str" files, the **structgen** program generates ".h" files (with the actual C structure definitions) and where possible ".c" files containing the protocol operations.

Both objects and protocols may be either *local* or *global*. Local objects and protocols are those needed by a single program and which will not generally be useful outside that program. Any objects or protocols used by more than one program should be global. (We often refer to the global ones as *system objects* and *system protocols*.) Examples of local objects are the edge and

segment objects of the **render** program, and an example of a local protocol is the start protocol which **render** uses to place objects on the active list and start lists. Local objects currently have the restriction that they may not subscribe to text I/O since the parser for textual object forms is considered global. Binary I/O of local objects may be used with care for communication between programs or storage of intermediate results. Any programs which use Binary I/O of local objects to communicate should also use the same set of local object definitions to insure that the object tags agree as expected.

Several *master files* are used to keep track of all the global objects and protocols. Maintenance of objects and protocols is achieved in a nearly automatic fashion using makefile dependencies extensively. There is a system makefile which insures that all the system master files are up to date. Further, there is a simple way of declaring makefile dependencies in program makefiles for local objects and operations. This means that the local files are always automatically regenerated when necessary so that they stay synchronized with the system definitions.

In the C objects package, protocol operations can be invoked "generically" if desired. This means that given a pointer to any object, the operation can examine the object tag and figure out exactly which procedure to call to perform the operation for that object type. For example, in display programs it is often desirable to just transform all the objects in the input stream to screen space before doing any further processing. Although the specific transformation routines for surfaces, curves, polygons, and other objects are all different, the abstract notion of what the operation does is uniform. So a *dispatching* function can examine the object tag, decide which specific routine to call, and pass the arguments on. The dispatching mechanism allows each program to have a "personalized" dispatching table which contains only the entries the program specifically requests. Theoretically, this means that only the desired routines need to be loaded when the program is compiled. What happens in practice, though, is that all of our programs want to be able to perform storage management, binary I/O, and debugging on **any** object, so a large set is usually loaded.

The dispatching mechanism for generically calling routines is currently a lookup table indexed by a function identifier and the object tag. The tables are built automatically, and tailored to the individual application program. Attempts to invoke an operation which is not loaded for the object type it is to be applied to produce an error message naming the object type and the attempted operation and cause the program to exit.

Protocol operations in the C object management scheme follow a strict naming convention. Every protocol operation has a short (usually 2 or 3 character) abbreviation. This abbreviation is concatenated with an underscore to the object type name to generate the name for the specific routine which implements the protocol operation for a given object type. The suffix "_obj" is used for the dispatching routine which does the table lookup and calls the specific protocol routines. For example, the bounding box protocol uses abbreviation "bb", has specific routines like **bb_surface_obj** and **bb_poly_obj**, and has generic dispatcher **bb_obj**.

The system maintains an include file for both the objects and the operations known by the system. These files define external integer variables which during execution will contain unique tag values. Any local objects and operations are also assigned unique tags which follow the system ones. Thus there is no distinction between local and system operations and objects during the dispatching. The actual values of the tags are initialized in the individual object and operation lookup tables which are provided for each program which deals with objects.

The C objects package does not support shared data during execution. References to previously named objects in text files result in copying of the data when the objects are constructed.

There is not currently a mechanism for automatically generating the text form of an object, nor any way to automatically generate the Yacc and C code to read and print objects. For a description of what the current text form is, and what conventions have been adopted see the User's Manual

appendix on text file format. Defining the text form for a new object is usually a matter of modifying an existing text format and its support code in the parser and printing functions.

5.2 Using Objects in Application Programs

This section describes how to use the object management scheme in application programs.

5.2.1 Using System Objects and Protocols

Accessing system known objects with system provided protocol operations is very simple. All routines which reference these objects and operations must include "alpha_1.h".

If the protocol operations do not need to be called generically, the routines can just be named as direct subroutine calls and linked with the regular Alpha_1 library. Even if generic calls are used, if only the binary I/O, storage management, and debug protocol operations are used generically, then a default dispatching table is available in the Alpha_1 library, and no further work is needed.

If generic calling of some other protocol operations is desired, a table specification must be provided. The table specification consists of a list of the objects which will be needed and the protocols which each object will need to access. The name of the table specification file is in the form "*pgm*-specs" (e.g., render-specs). The file should contain a standard header comment, and usually a

```
#include "basic_ops"
```

line to get everything in the standard dispatching table. Then it contains lines of the form:

object protocol1 protocol2 ...

See section 5.3.4 [Specs Files], page 71 for more on the format of this file, or copy the style of an existing one.

The "makefile.spec" file should contain the line:

OPTABLE *pgm*

and should also name the file "*pgm*-optbl.c" in the **CFILES** list. The "*pgm*-optbl.c" file is generated automatically from the "*pgm*-specs" file provided.

5.2.2 Using Local Protocols

It is sometimes desirable to create local protocol operations which are of interest only to a particular program. Some guidelines for whether an operation should become a protocol operation are mentioned here.

1. If the operation can be applied only to one object type, then it should not be a protocol operation.
2. If the operation is not a fairly direct function of the object type, then it should not be a protocol.
3. If the operation lends itself to being called generically in the program, and needs to be easily extensible to new object types, then it is a good candidate for a local protocol operation.

The procedure for implementing a local protocol is to specify certain attributes of the operation in a local protocol master file similar to the system master file. The necessary support (e.g., dispatching)

for the protocol is then generated automatically. The local protocol master file is named "*pname*-ptcl.mstr", where *pname* is often the same as *pgm* for the "*pgm*-specs". See section 5.3.8 [Protocol Master], page 76 for a description of the format of this file.

Note that defining a local protocol only arranges the linkages for generic calling — you must provide the actual operations for each object you wish to use it on, and specify the dispatching table entries in the "pgm-specs" file.

The "makefile.spec" entries should be of the form:

> OPTABLE *pgm* -P *pname*
> LOCPTCLS *pname*

In addition to "*pgm*-optbl.c" the **CFILES** list should include "*pname*-disp.c" (a set of dispatching routines generated automatically by the above specification lines) if any of the local protocols are invoked generically.

5.2.3 Using Local Objects

Definition of local objects which may subscribe to system or local protocols is also supported. Local objects currently have the restriction that they may not subscribe to the Text IO protocol. If an object must be textually communicated between two programs, then it needs to be a system object rather than a local object. Communication of local objects between programs is possible (although not recommended) with Binary I/O as long as the programs share the same set of local objects.

Local objects are typically temporary attributes which are used by the program to do its work, or objects which are heavily used in the program. The automatic generation of routines for creating, copying, and freeing (the storage management protocol operations) of local objects often makes creating a local object an attractive option. These objects can then be handled in much the same way that the more widely used system objects are handled.

In order to create a local object, a structure specification of the object must be defined in a file named "*objname*.str" (e.g., box.str). The format of this file is described in str files (see section 5.3.1 [Str Files], page 67). When all the local ".str" files have been constructed, a local object master file named "*oname*-str.mstr" must be provided (where *oname* is often the same as *pgm* for the "*pgm*-specs" file). This file lists the path name (relative to the current directory) of each of the local ".str" files. The protocols which the object subscribes to must be specified in the "*pgm*-specs" file as always.

If the new objects subscribe to system protocols, the **structgen** program (see section 5.3.2 [Structgen], page 69) can be used to generate some of the required operations (storage management, binary I/O, and debug protocols). Other operations will have to be hand-coded.

The necessary "makefile.spec" lines would be:

> OPTABLE *pgm* -O *oname*
> LOCOBJS *oname*

or, if local protocols were also being used:

> OPTABLE *pgm* -O *oname* -P *pname*
> LOCPTCLS *pname*
> LOCOBJS *oname*

5.2.4 Adding New System Objects

Adding new objects to the set of globally known objects is a process which must be done carefully. Usually, the definition of the new object is not as difficult as getting all the temporary linkages set up so that it can be tested before the installation. This section describes what must be done to make a new object, how to test it, and how to install the changes.

The first step is to specify the object data structure in a ".str" file. During development, the ".str" file will reside on a working directory of $astr. See section 5.3.1 [Str Files], page 67 for a detailed discussion of the format of these files, or copy the style of one that is similar to your new object.

When the ".str" file is ready, use **structgen** to generate the ".h" file and the binary I/O, text I/O, storage management, and debug protocol operations. If your object file is "obj.str", then a normal sequence might be:

```
hgen obj
StrgMgmt
ogen obj
BinIO
ogen obj
TextIO
ogen obj
Debug
ogen obj
```

This generates "obj.h", "B_obj.c", "T_obj.c", "D_obj.c", and "S_obj.c". Generally, the binary I/O, debug, and storage management operations are correct at this point. If you do need to make modifications, follow the procedures for building patch files (see section 5.3.3 [Patch Files], page 70) to make the changes in such a way that they do not have to be merged in by hand when small changes are made to **structgen**.

The text I/O protocols are never correct, and we don't generate patch files because those are never machine generated after the first time. Decide what you want the text from to look like (mimic the style of existing objects), modify the printer function in the "T_" file to generate that form, and then modify the parser (the **conv** program) on $parsd to read that format. This is not at all trivial, and not easily described since **lex** and **yacc** (two powerful and quite complex Unix tools) are involved. Copy style of existing objects and don't hesitate to ask for help.

The ".h" file that **structgen** built should be put on a working directory of $ai, and the ".c" files shouuld go on a working directory of $alo1 or $alo2.

In order to test the changes to the parser and all the code generated so far, you will have to successfully build a working directory version of **conv**. Here is the procedure:

- Make a working directory version of "structs.master" (on $mstr) with the name of your new ".str" file in it. Edit a copy of the "makefile.spec" file, un-commenting the "I =" and "ISF =" lines in the **WORK_DIR** section of the file. Run make on the $mstr working directory to get new "objects.h" and "objs.master" files. You will also get new versions of the files on $aisf, for building dispatching tables.
- On your $ai working directory, make a symbolic link to "alpha_1.h" on $ai. (This insures that your new version of "objects.h" will be used rather than the installed one.)
- Compile the C files on your $alo1 or $alo2 working directory. Be sure to uncomment the **INEW** variable in your copy of the "makefile.spec" so the new include file will be found.
- On a working directory of $alo1, generate a new version of the default dispatching table. You will need to uncomment the **MSTR** variable in the **WORK_DIR** section of the file and add an

ISFNEW variable which references the working directory under $aisf. After **gen-makefile** and "make depend" are run, say "make generic-optbl.c" to get the new dispatching table. (You don't actually need this step for making **conv**, but you may need it if you want to build any other programs.)

- On the working directory of $parsd, you will need to uncomment the **INEW**, **MSTR**, and **ISFNEW** variables in the **WORK_DIR** section. You will need also need the **AL** variable that references "ALL_OFILES" set up to get to your working directory of $alo1 or $alo2, depending on where your new object ".c" files are. Finally, after **gen-makefile** and "make depend", you are ready to "make conv".

Once you succeed in building **conv**, you can begin the cycle of debugging your parser changes. You will have to create the sample test data by hand this time. The usual test sequence is:

```
conv test.a1 | deconvcmd
```

Be sure to set $cvdir to your working directory of $parsd so that you are running your new version of **conv** rather than the installed one. Once this looks ok visually, a more rigorous check is to **diff** the results against your original test data, or to feed the results back through again:

```
conv test.a1 | deconvcmd | convcmd | deconvcmd
```

This insures that the parser is really an inverse of the printer. At this point, you have tested not only the text I/O changes, but some of the storage management operations and the binary I/O operations for your new object.

Finish writing and testing whatever other code is need to support your new object in other system programs. When you are ready to install the objects, check in the following files **only** (all the others will be generated by the **makeall** program):

- the ".str" file on $astr
- the "structs.master" file on $mstr
- new "makefile.spec" on $alo1 or $alo2 that has entries to generate the C files
- patch files (if any) on $alo1 or $alo2
- any ".c" files on $alo1 or $alo2 that aren't built by structgen (including the "T_" file if there is one
- the changes to the parser on $parsd

5.2.5 Adding New System Protocols

Adding new protocols to the set of protocols known by the system is much less involved than adding new objects, although less is done automatically for you. New objects are added frequently; new protocols are added rarely. To decide whether a new protocol is necessary, use the guidelines listed above for local protocols (see section 5.2.2 [Using Local Protocols], page 63). In addition, the protocol needs to be a system (or globally known) protocol only if the operation is used by a variety of programs which (probably) do not reside on a single directory.

The procedure for adding a new system protocol is similar to that for using a local protocol. The system protocol master file resides on $mstr and is called "ptcl.master". The format is the same as for the local ones (see section 5.3.8 [Protocol Master], page 76). You will have to provide the operations in the protocol for all existing system objects which should subscribe to the protocol.

In order to test the new protocol on working subdirectories

- Make a working directory version of "ptcl.master" (on $mstr) with the new protocol definitions in it. Edit a copy of the "makefile.spec" file, un-commenting the "I =" line in the **WORK_DIR** section of the file. Run make on the $mstr working directory to get new "operations.h" file.
- On your $ai working directory, make a symbolic link to "alpha_1.h" on $ai. (This insures that your new version of "operations.h" will be used rather than the installed one.)
- Compile any C files which implement the protocol operations on your $alo1 or $alo2 working directory. Be sure to uncomment the **INEW** variable in your copy of the "makefile.spec" so the new include file will be found.
- On a working directory of $alo1, generate a new version of the default dispatching table. You will need to uncomment the **MSTR** variable in the **WORK_DIR** section of the file. After **gen-makefile** and "make depend" are run, say "make generic-optbl.c" to get the new dispatching table.
- On the working directory where you have a program that uses the new protocol, you will need to uncomment the **INEW** and **MSTR** variables in the **WORK_DIR** section. You will need also need the **AL** variable that references "ALL_OFILES" set up to get to your working directory of $alo1 or $alo2, depending on where your new object ".c" files are. Finally, after **gen-makefile** and "make depend", you are ready to make the test version of the program.

5.3 Implementation Details

This section describes the special files used by the C object management scheme, the programs which maintain the objects and protocols, and the makefile entries which keep it all up to date. The first three sections describe how objects are defined in ".str" files, how those files are converted to appropriate ".c" and ".h" files by the **structgen** program, and how to build patch files if **structgen** doesn't get things quite right. The next two sections describe specifying and generating the dispatching tables which allow generic calls on protocol operations. Finally, the last few sections describe how the master lists of system objects and operations are maintained. The same mechanism is used (on a smaller scale) for local object and protocol lists.

5.3.1 Str Files

This section describes the format of a ".str" file. It may be helpful to compare a ".str" file in the system with its corresponding ".h" file. "Poly.str" is a simple example with several objects.

Like all Alpha_1 files, ".str" files should begin with a standard header. The following is an example of the required format for an object definition, extracted from "poly.str".

```
INCLUDE( poly )

OBJECT( vertex_obj )
subscribe( StrgMgmt, TextIO, BinIO, Debug, start )

  LINKS;
  ATTR_LIST;
  point_type pt;
```

The include list, with items separated by commas, specifies the names of ".h" files (without the ".h" extension) to be included in the C routines generated for the object. The name of the ".str"

file itself should always be specified. The file "alpha_1.h" is always automatically included and so need not be specified.

The "OBJECT" field gives the official struct name of the object, and by convention must end in "_obj".

The "subscribe" list specifies the protocols to which this object will subscribe. That is, it lists the groups of operations which will be supplied (either by automatic generation, or by hand) for the object. Only global protocols may appear, and the program that generates dispatching tables generates warning messages when requests are made that the object hasn't subscribed to. The warnings are generally ignored, and the program does the right thing.

There is a mechanism for specifying types, so that fields of objects may be declared to be of these types. (The parser assumes that any types it doesn't recognize are other objects, and generates generic function calls whenever it needs to deal with them). An example of the syntax is:

TYPE(int edge_code)

which says to treat anything of type edge_code as if it were an integer.

It is sometimes desirable to have lines in the ".str" file which are echoed into the ".h" file without any attempt at processing. The mechanism for accomplishing this is to put a percent (%) character at the beginning of each line which is to be passed straight through. You may also bracket a number of lines to be passed straight through by surrounding them with lines containing "%" and "%" at the beginning of the line. The most common use for this feature is to provide external declarations of functions that return the object types being defined, but that are not protocol operations.

There are three "kinds" of pointers in the ".str" files, and three symbols to represent them.

- Regular pointers usees it doesn't recognize are other objects, and gers will be followed by routines which copy, free, dump, etc. Objects pointed at by these are considered part of the original object.
- Reference pointers are specified with the greater-than (>) character. These pointers are considered just to be references. The objects they point to are not considered part of the object.
- List pointers are specified with the at (@) symbol. This is a regular pointer, except that the parser knows that it points at a list, and generates appropriate calls to list-handling routines when necessary. The load function in particular, checks that the objects in the list being loaded are all of the specified type.

The specification of the structure of the object is given almost exactly as in C. When the ".h" file is generated, the struct will look just like the one in the ".str" file except that there will be an extra field at the beginning which is for the unique object type tag. The special fields **LINKS** and **ATTR_LIST** will also be expanded in the ".h" file. **LINKS** specifies that the object will be able to be linked into lists with other objects. **ATTR_LIST** specifies that the object can have associated attributes.

An extension to the basic C-like syntax allows specification of variable length array elements. There may be a single variable length array in an object structure specification; it must appear as the last element in the specification. The size of the variable length array must be an integer-typed (i.e., int, short, long) field in the same structure. The constructor functions (new and init) for an object with a variable length element require that the size of the element be specified as an argument. Here is an example:

```
OBJECT( simple_polygon_obj )
subscribe( StrgMgmt, Debug )
```

```
LINKS;              /* might want to make lists of 'em */
ATTR_LIST;          /* for color, etc. */
int nvert;          /* Number of vertices in polygon */
point vert[nvert];        /* The vertices themselves */
```

Any structure element (except **LINKS** or **ATTR_LIST**) can have an initialization specified. Whenever a new element is created (by the "new_" function of the storage management protocol), the element will be initialized. All pointers and strings are initialized by default to NULL, but this may be overridden. Currently, only symbolic (e.g., defined symbols or global variables) and numeric (integer or floating point) values are supported. Initialization is specified by putting "= initial_value" after the element name. If a defined symbol or variable is used, it must be defined in one of the ".h" files in the **INCLUDE** list. Initialization of arrays works, all elements get the same value, but points can be initialized only by a global point variable.

The current view of the system is that in the object management system, the only required field in any object is the type tag. In order to be able to handle objects generically and not have to have procedure calls to get to the links and attribute lists of objects, the following two rules are enforced:

> If there are **LINKS**, that field must come first. If there is an **ATTR_LIST**, there must be LINKS, and the **ATTR_LIST** must immediately follow the **LINKS** field.

The **structgen** program checks these conditions, issues warning messages if the conventions are not followed, and generates as close to a correct output as it can.

Note that with this scheme, objects which are attributes themselves can be treated exactly like other objects. Any object can be an attribute of another one provided that it contains link fields.

Once the ".str" file is completed, its name must be listed in the master structs file (which is called "$mstr/structs.mstr" for the system objects). This file is just a list of the ".str" files which contain object specifications that the system is supposed to know about. The paths are relative to the $mstr directory.

5.3.2 Structgen

The **structgen** program works from the ".str"' file definitions of objects, and generates the appropriate C struct definitions in a ".h" file of the same name. The ".h" file also contains declarations of operations which are provided in the protocols to which the object subscribes.

The same program, with different flags, can also be run to generate some of the operations for the object. The program still works from the ".str" file specification, but this time generates a ".c" file which contains at least the templates for some of the system operations which the object needs to provide. It actually writes the routines for many standard operations, but these should be carefully checked by the user. Any routines which are not generated automatically must be filled in by hand by the user. The **structgen** program must be run once for each protocol which the object subscribes to; this is facilitated to a great extent by t have to have procedure calls to get to the linNING: Care must be taken not to regenerate the ".c" files from the struct definitions if modifications have been made by hand. A semi-automatic way of maintaining these hand-tailored changes with patch files is describe in the next section.

The commands for running the **structgen** program are loaded using the "sgen" keyword to **getcmds**, as in "getcmds sgen". All the commands assume that an object (or objects) has been defined in

a ".str" file using the prescribed format (see section 5.3.1 [Str Files], page 67). Suppose for this discussion that we have defined a set of objects in file "obj.str".

The ".str" file will normally reside on $astr for system objects, but the commands use the variable **$strdata** to decide where to access the data. The **$strdata** variable defaults to $astr. All the **structgen** commands are invoked by just naming the first part of the ".str" file, as in "hgen obj".

The commands are:

gen doesn't generate any code, just checks that the input file is parseable.

hgen generates the ".h" file corresponding to the ".str" file. (i.e., running "hgen obj" generates "obj.h" on the currently connected directory.)

ogen generates a ".c" file for the currently specified protocol.

Protocol flags are used by ogen to generate the desired ".c" files. The protocol flags may be set using the commands listed below. The lists in parentheses are the abbreviations for each operation in the protocol. The letters in braces are the identifiers for the protocols. This letter is used as a prefix for the ".c" file name. So "ogen obj" with **BinIO** set will generate "B_obj.c" containing binary load and dump routines for the objects defined in the ".str" file.

StrgMgmt Storage Management (new, mk, fr, and cp) S

TextIO Text Input/Output (rd and pr; rd is a dummy) T

BinIO Binary Input/Output (ld and dp) B

RasterDisp
Raster Display (rdisp; template only) R

LineDisp Line Display (ldisp; template only) L

Debug Debug Printing (dbg) D

StrgMgmt is the default. Exactly one of these flags can be turned on at one time.

The ".c" files generated by ogen are preceded by the letter(s) which follow the flag definitions above. For example, with **StrgMgmt** turned on, "ogen obj" will generate "S_obj.c" with new, make, free, copy, and other storage management routines for each of the objects defined in "obj.str".

5.3.3 Patch Files

The **structgen** program does a very good job of generating code for the storage management, binary I/O, and debug protocols. It will, however, still be necessary to modify the generated code on occasion. If at all possible, makefile entries should be added which regenerate the "B_", "S_", and "D_" files whenever the ".str" file or **structgen** itself changes. These entries look like

```
S_attr.c: $(ASTR)/attr.str $(ASTR)/attr.S $(STRGMGMT_GEN)
    structgen $(ASTR)/attr.str -o -S >S_attr.c
```

The **structgen** program attempts to merge in any hand-made changes by looking for a "patch" file on the same directory where the ".str" files reside. The patch files have the same name as the ".str" file, except that the suffix is "B", "C", "D", "H", "L", "M", "N", "R", "S", or "T", depending on which protocol operations the patches are for.

Generating the patch files is quite simple:

- Run **structgen** to get the default file.
- Copy the file "*name*.str" to "*name*.new".

- Edit "*name*.new" by hand to add the necessary changes. Each changed line should be commented with the string "/* PATCH */".
- Run a context diff with

  diff -c *name*.**str** *name*.**new** > *name*.*X*

  where "*name*.*X*" is named as described above.
- Run **structgen** again. The new output should be the same as your hand-edited version.

Don't forget to install the patch files when you install the ".str" files.

5.3.4 Specs Files

These files are, in simplest form, just lists of desired objects and the protocols which will be used to operate on them. Each object and the requested protocols for that object should be on a separate line. The first field of each line is the official name of the object (the name used for the struct in C programs that use the object). The remaining fields are just the names of the desired protocols, separated by spaces. An example line might be:

```
vertex StrgMgmt TextIO
```

The **mk-op-table** program, which turns "-specs" files into dispatching tables, detects all kinds of errors, including unrecognized names (usually misspellings) of objects or operations and illegal requests for operations to which an object does not subscribe. It is legal to request operations for an object on a number of lines of the specification file, and this is often used to group requests for readability.

In practice, this simple format of a line for each object turns out to be (in light of the large number of objects in the Alpha_1 system) tedious and difficult to maintain. Therefore, any C-preprocessor features may be used to make the file more manageable. These features include C-style comments, defined constants and macros, and include files.

A small set of include files have been provided on the $aisf directory, which is searched by **mk-op-table** for included files. These files insure that every system object which subscribes to a particular protocol is named in the specification file. Those provided currently are listed below. The "basic_ops" file includes all the others except **TextIO**, and is the usual one to include.

#include <StrgMgmt>
#include <BinIO>
#include <Debug>
#include <TextIO>
#include <basic_ops>

Below is a sample set of specifications. The files should normally begin with a standard header, but **be sure** to delete the "rcs_ident" line which appears outside the comment.

```
/* Get all the standard operations for all system objects. */
#include <basic_ops>

/* Define some local groups of protocols that will be needed. */
#define DISPLAY RasterDisp BoundingBox start
#define INTERNAL Debug StrgMgmt

/* These are the objects that can actually be displayed. */
subscribe( polygon, DISPLAY )
subscribe( line, DISPLAY )
```

```
/* These are displayed, but are internal objects (can't appear in
 * the input file - so no BinIO).
 */
subscribe( edge, DISPLAY )
subscribe( edge, INTERNAL )

/* Some attributes that are only used internally, can't appear in
 * the input.
 */
subscribe( bounding_box, INTERNAL )
subscribe( edge_shades, INTERNAL )
subscribe( edge_norms, INTERNAL )

/* Environment setting objects subscribe to the local start
 * protocol but not to the rest of DISPLAY.
 */
subscribe( trn_mat, start )
subscribe( render_mark, start )
subscribe( pop_mark, start )
subscribe( blinn_params, start )
```

5.3.5 Mk-op-table

Mk-op-table generates the dispatching table for a particular program based on the protocols requested in the "*pgm*-specs" file. The preparation of the "*pgm*-specs" file was discussed in Using System Objects and Protocols (see section 5.2.1 [Using System Objects and Protocols], page 63), and is also described in more detail in Specs Files (see section 5.3.4 [Specs Files], page 71).

The **mk-op-table** program uses the "*pgm*-specs" file, together with the system files "objs.master" and "ptcl.master" to generate a ".c" file which declares and initializes the one-dimensional array "optable". The user program should never explicitly reference this table, but should instead use the macros defined in "operations.h". The optable is conceptually a two-dimensional array, but is declared to be a vector, with the indexing done manually by the dispatching routines. The reasons for the manual indexing are related to extensibility of the table without impacting existing programs. The two dimensional array contains a row for each object (system or local) and a column for each operation (system or local). Each element of the table contains either NULL or the address of the routine which performs that operation for that object. The NULL entries occur for operations which are not members of a protocol which was specified for the object in the table specification file.

The optable file should be compiled and loaded with the main program for which it was generated. Since it references all the operations specified in the user's table specification file, including this file during loading will cause the appropriate object operations to also be loaded.

mk-op-table *pgm*-specs *oname*-objs.mstr *pname*-ptcl.mstr *pgm*-optbl.c

Output Construct a C source file defining the optable.

pgm-specs Specification of which operations are to be loaded for each object.

oname-objs.mstr
Local objects master file, generated by **mkobjs**.

pname-ptcl.mstr
Local protocol master file.

pgm-optbl.c
Name of the output file.

If no local objects are being used in the program, then the second argument to **mk-op-table** may be given as "-". The third argument may similarly be given as "-" if no local protocols are being used. So a common calling sequence is:

```
mk-op-table pgm-specs - - pgm-optbl.c
```

Calls to **mk-op-table** are only generated by the makefiles. New dispatching tables ("-optbl.c" files) need to be regenerated by the **mk-op-table** program whenever changes are made to one of the following:

- the master list of system objects
- The master list of system protocols
- the master list of local objects (if any)
- the master list of local protocols (if any)
- the specification ("-specs") file

The "makefile.spec" line:

```
OPTABLE pgm -O oname -P pname
```

expands in the makefile to:

```
pgm-optbl.c: $(MSTR)/objs.master $(MSTR)/ptcl.master \
                oname-objs.mstr pname-ptcl.mstr pgm-specs
        -cp pgm-optbl.c pgm-optbl.old
        mk-op-table pgm-specs oname-objs.mstr pname-ptcl.mstr \
                pgm-optbl.c
```

while the line

```
OPTABLE pgm
```

expands to

```
pgm-optbl.c: $(MSTR)/objs.master $(MSTR)/ptcl.master
        -cp pgm-optbl.c pgm-optbl.old
        mk-op-table pgm-specs - - pgm-optbl.c
```

The **mk-op-table** program often generates lots of warning messages about objects that don't subscribe to requested protocols, but it generates correct references to them in the resulting table. The messages are annoying, but it has no effect on the correct operation of the program.

5.3.6 Mkobjs

Mkobjs is used to add new objects to the system or allow an existing object to subscribe to a new protocol. There is some preparation involved before the program can be executed to install the object.

The user must specify the structure of the new object, as well as the protocols to which it subscribes. This is done via a .str file which has a format described in str files below (see section 5.3.1 [Str Files], page 67). Currently, all the system ".str" files reside on a common directory "($astr)". Local ".str" files should reside with the program which uses the objects.

Once the ".str" file for the new object has been prepared, its name (with relative path) must be added to the file containing the list of known objects. If the object is being added to the system-wide set of known objects, then the file "$mstr/structs.master" contains this list. If the object is being added locally, then there will be a local "*oname*-str.mstr" file for the program being created.

The **mkobjs** program extracts the names of the various objects from the named ".str" files, and generates an "*oname*-objs.h" file and an "*oname*-objs.mstr" file. The ".h" file may be installed on the standard include directory. The system version of "objects.h" contains the definition of the "generic object" which allows routines to generically access elements in the header fields. It also contains the declarations of the variables which will hold the unique object tags for the system. Local "*oname*-objs.h" files contain the tags for the local objects. The dispatching tables will arrange for the local tag numbering to begin immediately after the global tags. Any particular routine which uses objects should include at most one local "*oname*-objs.h" file. The "*oname*-objs.mstr" files contain a list of the objects and the protocols they subscribe to. This file and the system one ("objs.mstr") are used by **mk-op-table** to generate the dispatching table.

mkobjs *oname*-str.mstr *oname*-objs.h *oname*-objs.mstr [**-S**]

Output Creates an object master file and include file.

oname-str.mstr
: The list of ".str" files.

oname-objs.h
: Name of the ".h" file to be generated.

oname-objs.mstr
: Name of the object master file to be generated.

-S
: If specified, the program is being executed on the system (global) structs.

The "*oname*-objs.h" and "*oname*-objs.mstr" files need to be rebuilt whenever the local or system list of ".str" files changes, but the makefiles take care of this. The "makefile.spec" line:

LOCOBJS oname

expands in the makefile to:

```
oname-objs.mstr oname-objs.h: oname-str.mstr $(MSTR)/objs.master
        -cp oname-objs.mstr oname-objs.old
        mkobjs oname-str.mstr new-objs.h oname-objs.mstr
        -cp oname-objs.h oname-objs.h.old
        mv new-objs.h oname-objs.h
        touch .oname-objs.t
```

5.3.7 Mkptcls

Mkptcls is used for adding new protocol operations. The arguments to **mkptcls** are a set of three filenames, all required. The first file is the input file, a protocol master file, which specifies the operations. The other two, "*pname*-ops.h" and "*pname*-disp.c" will have their contents filled in by **mkptcls**.

mkptcls *pname*-ptcl.mstr *pname*-ops.h *pname*-disp.c [-S]

Output Build a protocol include file and dispatchers file.

pname-ptcl.mstr
: The list of protocol definitions.

pname-ops.h
The name of the protocol include file to be generated.

pname-disp.c
The name of the dispatchers file.

-S
If specified, the program is being run for the system (global) protocols.

The "*pname*-disp.c" file contains the dispatching routines which perform the indexing into the object/operation table, check that requested operations are loaded, and perform the calls on the requested object operations. These C routines are invoked by macros defined in "*pname*-ops.h" and should never need to be directly referenced in user programs. **BUG**: The argument list in the protocol master file does not currently have a facility for specifying the type of the arguments, and so the dispatchers have all the arguments declared to be pointers.

The "*pname*-ops.h" file which is built by **mkptcls** contains the dispatching macros which programs should use to invoke the dispatchers. The macros provide type-casting of any return values, and call the C subroutine dispatchers with the appropriate arguments. The file also contains declarations for the external variables which uniquely identify each operation known by the system. These variables are normally used only in the actual dispatching routines. The dispatching macros are named by appending "_obj" to the operation abbreviation as specified in the protocol master file. (e.g., If you specify an operation called "oper" with abbreviation "op", then it can be invoked generically with a call to "op_obj").

The arguments to the macro depend on the operation. If a **RETURNS** value is specified, then the macro will have to cast the result, since all the operations return a nondescript "address" type which is unlikely to match what the calling program expects back. Hence, the first argument to the dispatching macro will be the type of the pointer which is expected to be returned. If no arguments are given for the operation, then the macro will have to be explicitly given the type tag so that the dispatching can be performed. If arguments are specified, the macro assumes that the first one will be a pointer to the object which is to be operated on, and it finds the typetag from that value.

The **mkptcls** program can be used to create local protocol operations, or to add protocol operations to the set of those known by the system. The optional -S flag specifies that the system version is being updated, in which case the first argument should be "ptcl.master" which resides on $mstr and contains the list of system known protocols.

To create local protocol operations, build a local protocol master file which contains the specifications for the new operations. The format of the protocol master file is described in protocol master (see section 5.3.8 [Protocol Master], page 76). Be sure that local operation names and abbreviations do not collide with those known by the system. When building a local protocol it may also be useful to examine the system master file for examples of operations which are similar to the local ones being created.

Like the other maintenance programs for object management in C, **mkptcls** is never called except from a makefile. The "*pname*-ops.h" and "*pname*-disp.c" files need to be regenerated whenever the local or global protocol master lists are changed. The "makefile.spec" line:

LOCPTCLS pname

will expand in the makefile to:

```
pname-ops.h pname-disp.c: pname-ptcl.mstr $(MSTR)/ptcl.master
        -cp pname-disp.c pname-disp.old
        mkptcls pname-ptcl.mstr new-ops.h pname-disp.c
        -cp pname-ops.h pname-ops.h.old
        mv new-ops.h pname-ops.h
        touch .pname-ops.t
```

5.3.8 Protocol Master

A protocol is defined by having at least one entry in the system "ptcl.mstr" file or a local "*pname*-ptcl.mstr" file which specifies an operation which is a member of the protocol. The text in this file may include any comments and explanations desired, provided that they do not begin with the "#PTCL" marker. To add new operations to the file, you may use the template provided at the front of the file:

    #PTCL() OP() RETURNS() ARGS()

All four fields must be present on each protocol line, even if some of the values remain null. The **PTCL** field gives the name of the protocol, while **OP** specifies a particular operation in the protocol. This specification is used to determine the macros for dispatching, i.e., calling object operations generically. So these options determine the calling sequence.

If a **RETURNS** value is specified, then the macro will have to cast the result, since all the operations return a nondescript "address" type which is unlikely to match what the calling program expects back. Hence, the first argument to the dispatching macro will be the type of the pointer which is expected to be returned. If the specified return type does not contain the string "type", it will be assumed that the operation returns a constant type, so the dispatching macro will not require a type argument.

If no arguments are given in the **ARGS** field, then the macro will have to be explicitly given the type tag so that the dispatching can be performed. If arguments are specified, the macro assumes that the first one will be a pointer to the object which is to be operated on, and it finds the typetag from that value.

Currently, all arguments are assumed to be pointers, and the dispatching routines (in "dispatchers.c") declare them that way. To do the declarations correctly involves building a mechanism for specifying types in the **ARGS** list.

The system protocols are found in the file "$mstr/ptcl.master" and are described in the section System Protocols List below (see section 5.4 [System Protocols List], page 76).

5.4 System Protocols List

This section contains a list of the currently implemented protocols in the Alpha_1 system. We are able to generate functions for the storage management, binary I/O, and Debug protocols directly from a the ".str" files.

Storage Management

The storage management protocol (StrgMgmt) contains the operations used to handle object instances in system memory. This protocol must be provided by all object types within the system, regardless of other protocols to which they may subscribe.

Operations are:

- Make (mk) allocates an instance and fills it in from parameters. The make operations will never be called generically, since the parameters vary with the datatype being created.
- New creates and returns a new instance, but does not fill it in. It does initialize the object type tag field, and sets all embedded links to null for safety. Initial values specified in the ".str" file are also filled in. This allows the storage to be allocated for variable size objects and then filled in by the creating code, in cases

where duplicate storage and copying would be involved, or where there are a large number of optional parameters which are usually defaulted.

- Free (fr) deallocates an instance, recursively killing any linked portions of the data structure. Degenerates to system "free" function if there are no links.
- Copy (cp) copies an instance, recursively copying embedded linked objects. It degenerates to the system "copy" if there are no links.
- Initialize (init) sets initial values in the given object instance. It is exactly like new, except that it does not allocate the storage. In fact, the new operation uses the initialize operation.
- Fill is like make, but loads the given values into the given object instance. Make is actually implemented as new followed by fill.
- List predicate (links) returns TRUE if the object type has list links and FALSE if it doesn't.
- Attribute predicate (attrs) returns TRUE if the object type may have an attribute list, and FALSE if not.

Text IO

The Text IO protocol (TextIO) includes read (rd) and print (pr) operations.

The object management system has a convenient text form (allowing, among other things, comments and named objects for a form of sharing data) which is interpreted by a parser written in Yacc. The binary files still exist for efficient communication between programs, but will be unseen by the user in general. There are also corresponding "pretty-printers" which generate the human-readable text form when presented with the object.

Currently, the read operations are dummy routines (their semantics are contained in the parser). They may become more real if we switch to a simplified text form that doesn't require a parser.

Binary IO

The Binary IO protocol (BinIO) includes dump (dp) and load (ld) operations.

The system dumps and loads in a "block transfer" mode, with the unique integer type tag and the allocated size as the first data items to be written/read. All datatype instances contain their integer object id, which is used to index into generic operation tables. The load routines fill in the pointers to the allocated locations.

Raster Display

The raster display protocol (RasterDisp) includes raster display (rdisp) and remove predicate (p_rem) operations.

The objects providing raster rendering operations break down into two groups: the ones directly renderable, and the ones which aren't but convert to something which is. The remove predicate decides whether the object has expired (in the scanline-ordered rendering sense).

Line Display

The line display protocol (LineDisp) has just the line display (ldisp) operation.

The view program currently does this operation in a somewhat ad hoc way. In general, there needs to be a way of interpreting the geometry of designed objects as a set of line segments which can be drawn.

Bounding Box

The bounding box protocol (BoundingBox) has only the bounding box (bb) operation.

This protocol is used by operations in both the **view** and **render** programs. The bounding box information is stored as an attribute of the renderable object. The routines which try to access the bounding box information can call the bounding box routine if the information is not already present.

Debug

The debug protocol (Debug) has one operation (dbg) which prints a debugging (verbose) form of the object. The output is not readable by any program, but is extremely valuable for use in dbx. The object is printed with information normally not shown, such as pointer values and its own address. These routines are generated completely automatically.

MAP

The map protocol (MAP) has two operations which push objects through transformation matrices. One operation (map_tran) requires specification of the transformation matrix, while the other (map) allows simply an indication of the desired space of the result (e.g., screen space).

6. C Packages

6.1 C Package Introduction

Packages in the C code of Alpha_1 are groups of defined types, defined constants, macros, and subroutines. Examples of existing packages in the C code include handling for doubly-linked lists, common geometric operations on points and vectors, and B-spline surface subdivision and refinement.

Typically, a package consists of a single include (".h") file and several C source (".c") files. The include file includes:

- type definitions which are specific to the package
- relevant defined constants
- macros used within the package
- (external) declarations of any global variables associated with the package
- (external) declarations of all subroutines in the package defining their return values (if any). Comments surrounding the declaration give a short description of the routine, its calling sequence and a description of the arguments.

The global variables (if any — their use is discouraged) are declared (and storage space allocated) in exactly one of the package ".c" files. Usually this file contains only variable declarations and no subroutines, and has the same name as the ".h" file in which the variables are declared external.

Everything — typedefs, defined constants, global variables, macros, and subroutines — should be tagged, using a comment in the style described in major comments (see section 3.4.6 [Major Comments], page 19). The use of the tags database is described in the chapter on getting started (see chapter 2 [Getting Started], page 5) in Alpha_1. Note that subroutines are tagged twice: once during external declaration in the ".h" file and once in the ".c" file. The tags program comes up with the ".h" reference first, which is often sufficient information, and the ".c" reference afterwards for more in-depth information.

There are conventions for naming macros, types, and subroutines; these are described in the section on naming conventions (see section 3.4.4 [Naming Conventions], page 17).

Packages are always included in a library, so that programs which use the package can load just what they need. The ".h" file should reside on $ai or an appropriate subdirectory, while the ".c" source files reside on an appropriate subdirectory of $srclib. When developing a package, consult with a member of the support staff to determine which existing library should include your package, or whether a new library is required. The makefiles on the library directory group the files according to their packages (see section 3.6 [Make], page 22).

The remaining sections of this chapter describe particular packages which are available and used throughout the Alpha_1 C code. Programmers on Alpha_1 should be familiar with what packages are available. This documentation can be used as a first reference, with the tags facility providing more detailed information when necessary.

In the sections below, subroutines and macros are documented in a similar style. The naming conventions usually will indicate which are macros (their names are usually upper case).

6.2 Alpha_1 Miscellaneous Package

This section describes a set of definitions and routines which are not really a package, but which form a basic underlying extension to the C language and are used throughout Alpha_1 C code. The file "misc.h" contains these definitions; any subroutines are found in the standard Alpha_1 library (libA1.a).

The following low-level type definitions are provided:

string	A pointer to a character, for character strings.
boolean	For data which has only TRUE or FALSE values.
fn	The name is a function which returns an integer. This is probably only useful for extern declarations.
byte	The size of a single character.
address	Just a pointer — used by routines that don't care what kind of object it points at (e.g., the allocation routines).

The following two macros compute the maximum or minimum of two numeric values:

MAX(a, *b*)

Returns	<number> Maximum of a and *b*.
a, *b*	<number> The two numbers to compare.

MIN(a, *b*)

Returns	<number> Minimum of a and *b*.
a, *b*	<number> The two numbers to compare.

The **BOUND** macro bounds a value against the two given extremes.

BOUND(*lo*, x, *hi*)

Returns	<number> A number in the range *lo* to *hi*.
lo, *hi*	<number> The bounds of the range.
x	<number> The value to be "clipped" against the bounds.

The **SWAP** macro exchanges the values of two variables of the specified type.

SWAP(*type*, *x*, *y*)

Returns	<any> Exchange *x* and *y*.
type	<typename> The type of the arguments, for declaring a temporary storage location.
x, *y*	<any The values to be exchanged.

Some useful constants are:

TRUE	A number that C recognizes as a "true" value.
FALSE	A number that C recognizes as a "false" value.
INFINITY	A very large floating point number.
BADINT	An unacceptable integer return value.
EPS	Floating point epsilon (general accuracy of floating point operations).

Two macros are useful for floating point "equality" comparisons: **APX_EQ** returns a boolean value

indicating whether the floats are approximately equal (using EPS), and **REL_EQ** which returns a boolean value indicating relative equality. It scales EPS by the size of the values before using it for comparison.

APX_EQ(*x*, *y*)

Returns <boolean> **TRUE** if the two numbers are approximately equal.
x, *y* <number> The numbers to compare.

REL_EQ(*x*, *y*)

Returns <boolean> **TRUE** if the numbers are relatively equal. The value of epsilon is scaled by the larger of the two numbers before the comparison.
x, *y* <number> The numbers to be compared.

Errors which can't be recovered from usually print an error message and then exit. The **SCREAM** macro prints the message to standard error.

SCREAM((*msg*, *val1*, ...))

Returns <none> Print a message to the standard error output.
msg <string> Format string just like **printf**.
val1, ... <any> Values referenced in the format string.

A double set of parenthesis is required, as in

```
SCREAM(("Variable foo has invalid value %d%n.", foo ));
```

To cause the program (leaving a core dump for debugging) the macro **DIE** is used. It is usually preceded by a message from **SCREAM**.

DIE

Returns <none> Aborts the program.

Storage allocation routines are accessed using one of the macros **NEW** or **NEW_1**.

NEW(*type*,*size*)

Returns <type *> A pointer to a newly allocated block, cast to the given type.
type <typename> The type of the pointer for casting of the result.
size <int> Size of the block to allocate.

NEW_1(*type*)

Returns <type *> A pointer to a newly allocated block, cast to the given type with size determined as size of the type.
type <typename> The type of the pointer for casting of the result.

The latter assumes that the type requires a fixed amount of storage which can be determined using the standard C function "sizeof". Examples of types which are not fixed size are those which contain embedded variable size arrays. Both macros **SCREAM** and **DIE** if there is an error during allocation of the requested storage.

Releasing allocated storage when you are finished using it is accomplished with the **dispose** macro:

dispose(*ptr*)

Returns <none> Frees the block of storage pointed to by *ptr*.
ptr <pointer> The pointer to the block of storage to be freed.

(The upper-cased usage is prefered; the lower case is for compatability with old code. Note that you cannot call the C **free** function directly from code that has these macros defined. The **FREE** macros are companions to **NEW** and **NEW_1**, and eventually call the C **free**, just as **NEW** eventually calls the C **malloc** function.

Occasionally it is desireable to zero an allocated area using

ZAP(*ptr*)

Returns	<...> ...
ptr	<...> ...

To copy an allocated area, the most low-level routine is

COPY(*type*, *ptr*)

Returns	<...> ...
type	<...> ...
ptr	<...> ...

which returns a pointer to the new copy. Note that most system objects have their own copy function which traces and copies data referenced by pointer as well (see section 6.7 [Protocol Support], page 95).

To extend an allocated area (usually for a type with variable size), use

EXPAND(*type*, *ptr*, *extra*)

Returns	<...> ...
type	<...> Type of the area.
ptr	<...> Original pointer.
extra	<...> Number of additional bytes needed.

The macro

SIZE(*ptr*)

Returns	<...> ...
ptr	<...> ...

returns the size of the allocated area to the next larger power of 2.

There are some very basic point data structures and operations defined. A more complete points package is described later in Points Package (see section 6.4 [Points Package], page 86). The two basic datatypes are

point
Hpoint

for 3D euclidean and homogeneous points respectively. Macros exist for shorthand accessing of both types of points:

X, Y, Z
HX, HY, HZ, HW

These macros are used as structure references (e.g., ptr->X or pt.X).

The most basic point operations are:

dot_prod(*p1*, *p2*)

Returns <...> ...
p1, p2 <...> Pointers to points.

for computing dot products,

length(*pt*)

Returns <...> ...
pt <...> Pointer to a point.

for lengths of vectors, and

cross_prod(*p1, p2*)

Returns <...> ...
p1, p2 <...> Pointers to points.

for cross products. **Dot_prod** is a macro; the other two are subroutines.

A few indentation utilities help in printing structured text (especially the printing operations of the Text IO and Debug protocols).

INDENT(*lvl*)

Returns <...> ...
lvl <...> ...

EINDENT(*lvl*)

Returns <...> ...
lvl <...> ...

FINDENT(*file, lvl*)

Returns <...> ...
file <...> ...
lvl <...> ...

The level argument indicates how many four-space levels to indent. The macros apply the indentation to the standard output, standard error, and a specified file, respectively.

Used in much of the user-defined attribute processing,

mk_name(*string*)

Returns <...> ...
string <...> ...

calls the standard "strcopy" routine, but also handles storage allocation for the new copy.

6.3 List Package

The list package provides macros and routines for manipulating lists of objects in C. Any object with "link" fields can be an element of a list, and operated on by these routines. The

TLISTLINKS(*type*)

Returns <...> ...
type <...> ...

macro is used in a struct definition to create the link fields of an object.

To access the previous and next elements in the list, use

P(*obj*)

Returns	<...> ...
obj	<...> ...

and

N(*obj*)

Returns	<...> ...
obj	<...> ...

respectively.

The basic list macros for adding elements to a list are:

ADD(*new*, *first*)

Returns	<...> ...
new	<...> ...
first	<...> ...

INSERT(*new*, *after*)

Returns	<...> ...
new	<...> ...
after	<...> ...

The new element is put at the beginning of the list with **ADD**, and after the given element with **INSERT**.

To unlink an element from a list, you can use

REMOVE(*element*)

Returns	<...> ...
element	<...> ...

although

DEL(*element*, *base*)

Returns	<...> ...
element	<...> ...
base	<...> ...

is safer because it makes sure the the pointer to the beginning of the list is also updated in case the removed element was the first one in the list.

Other useful list operations are

REVERSE(*list*)

Returns	<...> ...
list	<...> ...

to reverse a list, and

APPEND(*list1*, *list2*)

Returns <...> ...
list1 <...> ...
list2 <...> ...

to append two lists. A safer version of **APPEND** is

APPEND_TO(*type*, *list1*, *list2*)

Returns <...> ...
type <...> ...
list1 <...> ...
list2 <...> ...

which insures that the value of list1 is the result of the append (which might not be true with **APPEND** if list1 was NULL). A similar operation which puts the result in the second list argument is

PREPEND_TO(*type*, *list1*, *list2*)

Returns <...> ...
type <...> ...
list1 <...> ...
list2 <...> ...

There are some other routines to make life easier in dealing with lists.

TRACE(*loopvar*, *initvalue*)

Returns <...> ...
loopvar <...> ...
initvalue <...> ...

sets up a for loop construct which walks the list. Typical usage is

```
TRACE( element_ptr, list )

    /* Do something with each element in element_ptr. */
```

To free all the elements in a list,

FREE_LIST(*list*)

Returns <...> ...
list <...> ...

may be used, although for system objects which subscribe to the storage management protocol (see chapter 5 [Object Management in C], page 61) the **fr_obj_list** routine may be more appropriate.

Predicates for testing whether an element is the first or last in a list are

FIRSTP(*element*)

Returns <...> ...
element <...> ...

ENDP(*element*)

Returns <...> ...
element <...> ...

respectively.

Circularly linked lists are also supported. To create one,

MK_CIRCULAR(*element*)

Returns <...> ...
element <...> ...

makes a single element into a circular list. Then **INSERT** works to keep it a circular list while adding new elements. To break a circularly linked list into a linear one,

BREAK_CIRCULAR(*element*)

Returns <...> ...
element <...> ...

is used. The given element becomes the beginning of the list.

Analogous to TRACE,

TRACE_CIRCULAR(*loopvar*, *initvalue*)

Returns <...> ...
loopvar <...> ...
initvalue <...> ...

traverses a circular list. It also works on linear lists.

6.4 Points Package

The points package provides a large number of operations on points. The package defines a 3-coordinate point, and some of the routines work on E3 points while others expect projective 2D (P2) points. A P2 point is structurally just like an E3 point but the third coordinate is interpreted as a W value instead of a third dimension. E3 points can also be considered as vectors in 3-space (with Z=0). (This is, of course, really sleazy. See the points and vectors package described in the User's Manual for a more "type-conscious" implementation.) The **point_obj** package provides various types of points for more flexibility within surface control meshes and other aggregates.

The point datatype is

point

Macros exist for shorthand accessing of the fields:

HX, HY, HZ, HW

These macros are used as structure references (e.g., ptr->X or pt.X).

To create points, use

mk_point(*x*, *y*, *z*)

Returns <...> ...
x <...> ...
y <...> ...
z <...> ...

The **cross_prod**, **dot_prod**, and **length** routines were described previously in the section on the Alpha_1 Miscellaneous Package.

Coordinate-wise addition and subtraction of points is via

pt_add(*result*, *p1*, *p2*)

Returns	<...> ...
result	<...> ...
p1	<...> ...
p2	<...> ...

pt_sub(*result*, *p1*, *p2*)

Returns	<...> ...
result	<...> ...
p1	<...> ...
p2	<...> ...

All the C point routines expect the storage for the result to be provided as an argument, and all arguments are **pointers to** points, rather than actual point structs. The result pointer may be one of the argument pointers if the values are to be overwritten. It may also be NULL, although this will often be unsafe. If NULL is given as the result pointer, the result is stored in a static area within the function and a pointer to that area is returned. Note that this area will be overwritten at the next call, so use this option carefully.

Routines for multiplying points include

pt_mul(*result*, *pt*, *scalar*)

Returns	<...> ...
result	<...> ...
pt	<...> ...
scalar	<...> ...

for multiplying by a constant and

pt_mat_mul(*result*, *pt*, *mat*)

Returns	<...> ...
result	<...> ...
pt	<...> ...
mat	<...> ...

for multiplying a point through a 3x3 matrix. Use **pt_map** for multiplying through a 4x4 transformation matrix.

The

pt_normalize(*result*, *pt*)

Returns	<...> ...
result	<...> ...
pt	<...> ...

routine normalizes a vector, dividing through by its length.

The

zerovec(*pt*)

Returns <...> ...
pt <...> ...

predicate returns a TRUE value if the vector is identically zero.

The averaging routine

pt_interp(*result*, *p0*, *p1*, *t*)

Returns <...> ...
result <...> ...
p0 <...> ...
p1 <...> ...
t <...> ...

finds a point at a specified percentage of the way between the two given points (i.e., on the line between them).

pt_offset(*result*, *base_pt*, *dir_pt*, *len*)

Returns <...> ...
result <...> ...
base_pt <...> ...
dir_pt <...> ...
len <...> ...

constructs a new point which is offset the given distance in the given direction from the base point.

pt_perp(*result*, *base_pt*, *p0*, *p1*, *len*)

Returns <...> ...
result <...> ...
base_pt <...> ...
p0 <...> ...
p1 <...> ...
len <...> ...

constructs a new point which is offset a given distance from a base point, perpendicular to a plane.

pt_parallel(*result*, *base_pt*, *p0*, *p1*, *len*)

Returns <...> ...
result <...> ...
base_pt <...> ...
p0 <...> ...
p1 <...> ...
len <...> ...

constructs a new point which is offset from a base point, parallel to a line through another pair of points.

Finally,

pt_flop(*result*, *p1*, *m*, *p2*)

Returns <...> ...

result <...> ...
p1 <...> ...
m <...> ...
p2 <...> ...

computes ax+b where a and b are points and x is a scalar.

Lines

The next set of routines expect P2 points or lines for their inputs. They are to be thought of as 2D operations, not 3D. Lines should be produced only by the "line" routines below.

There are several line construction routines:

line_thru_2pts(*result*, *p1*, *p2*)

Returns <...> ...
result <...> ...
p1 <...> ...
p2 <...> ...

line_pt_vec(*result*, *pt*, *vec*)

Returns <...> ...
result <...> ...
pt <...> ...
vec <...> ...

line_pt_angle(*result*, *pt*, *radian_angle*)

Returns <...> ...
result <...> ...
pt <...> ...
radian_angle
<...> ...

line_pt_parallel(*result*, *pt*, *line*)

Returns <...> ...
result <...> ...
pt <...> ...
line <...> ...

line_vertical(*result*, *xoffset*)

Returns <...> ...
result <...> ...
xoffset <...> ...

line_horizontal(*result*, *yoffset*)

Returns <...> ...
result <...> ...
yoffset <...> ...

Routines which extract information from lines are

dist_pt_line(*pt*, *line*)

Returns	<...> ...
pt	<...> ...
line	<...> ...

for computing the perpendicular distance from a point to a line,

pt_on_line(*result*, *line*, *dist*, *base*)

Returns	<...> ...
result	<...> ...
line	<...> ...
dist	<...> ...
base	<...> ...

for computing a point on a line a given distance from a base point on the line, and

dir_of_line(*result*, *line*)

Returns	<...> ...
result	<...> ...
line	<...> ...

dir_perp_line(*result*, *line*)

Returns	<...> ...
result	<...> ...
line	<...> ...

for computing the direction of a line and the perpendicular to the direction of a line respectively.

Arcs

These routines construct arcs, which are expected to eventually be installed as parts of curves or surfaces. An "arc" is just an array of 3 points.

The constructors are:

arc_tan_3lines(*result*, *line1*, *line2*, *line3*)

Returns	<...> ...
result	<...> ...
line1	<...> ...
line2	<...> ...
line3	<...> ...

arc_rad_tan_2lines(*result*, *radius*, *line1*, *line2*)

Returns	<...> ...
result	<...> ...
radius	<...> ...
line1	<...> ...
line2	<...> ...

arc_thru_3pts(*result*, *endpoint1*, *middlept*, *endpt2*)

Returns	<...> ...
result	<...> ...
endpoint1	<...> ...

middlept <...> ...
endpt2 <...> ...

arc_end_center_end(*result*, *endpt1*, *centerpt*, *endpt2*)

Returns <...> ...
result <...> ...
endpt1 <...> ...
centerpt <...> ...
endpt2 <...> ...

arc_end_corner_end(*result*, *endpt1*, *cornerpt*, *endpt2*)

Returns <...> ...
result <...> ...
endpt1 <...> ...
cornerpt <...> ...
endpt2 <...> ...

arc_pt_tan_line(*result*, *pt*, *tangent*, *line*)

Returns <...> ...
result <...> ...
pt <...> ...
tangent <...> ...
line <...> ...

Two routines for extracting information from arcs are provided:

radius_of_arc(*arc*)

Returns <...> ...
arc <...> ...

center_of_arc(*arc*)

Returns <...> ...
arc <...> ...

6.5 Transformation Package

The transformation package provides variables and utilities for setting up and maintaining an object transformation stack and a current perspective transformation. To get access to the global variables, files which use the transformation stack routines must include the file "scene.h".

The basic transformation datatype is a

trn_mat

There are also **push_mark** and **pop_mark** objects which trigger pushes and pops on the stack. The transformation stack handling usually occurs in a **read_obj** loop in the main procedure of a program.

These routines set up and maintain the object transformation stack and the perspective transformation. The appropriate start routine should be called whenever new transformations are to be applied.

mat_init()

Returns <...> [SR] Initializes the stack and perspective transformation.

is used to initialize the stack and perspective transformation.

To put a new object transformation on the stack or set the perspective transformation, use

start_tmat(*mat*)

Returns <...> ...
mat <...> ...

To pop or push the transformation stack, use

start_pop(*pop_mark*)

Returns <...> ...
pop_mark <...> ...

start_push(*push_mark*)

Returns <...> ...
push_mark <...> ...

These routines are used to create transformation matrices (and are used primarily by the textual input parser).

mk_scr_tr(*hither*, *yon*, *vwport*, *eye*, *screen*)

Returns <...> ...
hither <...> ...
yon <...> ...
vwport <...> ...
eye <...> ...
screen <...> ...

mk_scale(*magnitude*, *direction*)

Returns <...> ...
magnitude <...> ...
direction <...> ...

mk_translate(*magnitude*, *direction*)

Returns <...> ...
magnitude <...> ...
direction <...> ...

mk_rotate(*angle*, *deg_rad*, *axis*)

Returns <...> ...
angle <...> ...
deg_rad <...> ...
axis <...> ...

ident_mat(*mat*)

Returns	<...> ...
mat	<...> ...

Transformation matrices can be multiplied with

mult(*mat1*, *mat2*)

Returns	<...> ...
mat1	<...> ...
mat2	<...> ...

Some routines are provided for pushing objects through the transformations:

pt_map(*in_pt*, *trans*, *trans_pt*)

Returns	<...> ...
in_pt	<...> ...
trans	<...> ...
trans_pt	<...> ...

Hpt_map(*in_pt*, *trans*, *trans_pt*)

Returns	<...> ...
in_pt	<...> ...
trans	<...> ...
trans_pt	<...> ...

To do the homogeneous division of projective point, use

coerce_to_e3(*result*, *mat_elt_ptr*, *pt_type*)

Returns	<...> ...
result	<...> ...
mat_elt_ptr	
	<...> ...
pt_type	<...> ...

where **pt_type** is **E2**, **P2**, **E3**, or **P3**.

Other mapping functions are:

mesh_map(*pt_mat*, *trans*)

Returns	<...> ...
pt_mat	<...> ...
trans	<...> ...

Hmesh_map(*Hpt_mat*, *trans*)

Returns	<...> ...
Hpt_mat	<...> ...
trans	<...> ...

srf_map(*srf*, *trans*)

Returns	<...> ...
srf	<...> ...
trans	<...> ...

poly_map(*poly*, *trans*)

Returns	<...> ...
poly	<...> ...
trans	<...> ...

line_map(*line*, *trans*)

Returns	<...> ...
line	<...> ...
trans	<...> ...

6.6 List of System Objects

Certain objects in the Alpha_1 system all provide some operations which have identical calling sequences and whose semantics are clear. These operations are described in the next section on protocols. The objects for which some or all of these operations have been provided are listed below.

Each object has a unique numeric type tag which identifies all instances of the object, and is stored as a field in the instance. A set of global variables with names of the form

t_objname

is defined with values equal to the typetag of objname.

Each object also has a predicate of the form

is_objname_obj(*obj*)

Returns	<...> ...
obj	<...> ...

which returns a true value if *obj* is of type *objname*.

Objects in the system at the time of this writing include (for a complete and current list, see the file "objects.h"):

line	A line segment.
vertex	A 3D point.
contour	An ordered collection of vertices, may be circular.
polygon	A collection of (circular) contours and other attributes.
polyline	A collection of (linear) contours and other attributes.
crv_object	A curve with links and attributes.
srf_object	A surface with links and attributes.
uv_coord	Parametric (2D) values of a point.
shade	An intensity value.
normal	A surface normal.
rgb	A color.
opacity	A transparency value.
light_vec	A light vector position in 3-space.

blinn_params
: Blinn parameter values: specular, diffuse, index of refraction.

resolution
: A tolerance for flatness testing of surfaces.

width
: The width of a line or polyline (in pixels).

dist_params
: Facet distribution parameters for shading.

bbox
: A bounding box in 3-space.

render_mark
: Current scanline, for synchronizing subdivision and rendering processes.

push_mark
: Indicates a transformation stack push operation.

pop_mark
: Indicates a transformation stack pop operation.

trn_mat
: Transformation matrix: screen or object.

edge_flag
: Attribute of a surface, records straightness of edges.

surf_corners
: Attribute of a surface, records corner information.

subdiv_dir
: Attribute of a surface, indicates last subdivision direction.

string_attr
: User defined string attribute.

float_attr
: User defined float attribute.

integer_attr
: User defined integer attribute.

str_attr_hdr
: Internal object, header of list of string_attr.

fl_attr_hdr
: Internal object, header of list of float_attr.

int_attr_hdr
: Internal object, header of list of integer_attr.

6.7 Protocol Support

The following lists protocols (or families of operations) which are provided for objects in the Alpha_1 system. If requested, and if the object "subscribes" to the protocol, any of these operations can be performed for any object. For more information about protocols see chapter 5 [Object Management in C], page 61.

There is a convention for naming these routines. Every operation and every object has a short abbreviation; the name of the routine which performs that operation for that object is "operation-abbrev_object-abbrev" (e.g., cp_poly).

The storage management protocol has the following operations:

new
: (new) to allocate storage.

mk
: (make) to allocate and fill in storage.

fr
: (free) to recursively free storage.

cp (copy) to recursively copy storage.

The Text Input/Output protocol has operations:

rd (read) read object from character input.

pr (print) print object into 'pretty' character output.

The Binary Input/Output protocol has operations:

ld (load) read object from binary input

dp (dump) dump object into binary file

The bounding box protocol has a single operation which computes the bounding box of an object.

bb (boundingBox) compute a bounding box.

The raster display protocol has operations:

rdisp (raster display) display as shaded image, or beat into a more primitive object.

p_rem (remove object) predicate, returns true if object has expired.

The Linedrawing Display protocol has operation:

ldisp (line draw) display with vectors.

The Debug print protocol has one operation:

dbg (debug print) print the contents of the object on stderr.

The above protocol operations can be invoked "generically" using the routines below. The dispatching routines figure out what kind of object is to be operated on and call the appropriate subroutine. Exception: The mk protocol cannot be called generically because its arguments depend on the object.

To create an object, use

new_obj(*type*, *typetag*)

Returns	<...> ...
type	<...> ...
typetag	<...> ...

where a typical calling sequence might be

```
foo = new_obj( poly, t_polygon );
```

To free or copy an object, use

fr_obj(*ptr*)

Returns	<...> ...
ptr	<...> ...

cp_obj(*type*, *ptr*)

Returns	<...> ...
type	<...> ...
ptr	<...> ...

Objects print at a given indentation level:

pr_obj(*ptr*, *lvl*)

Returns	<...> ...
ptr	<...> ...
lvl	<...> ...

Although the routine

ld_obj(*type*, *typetag*)

Returns	<...> ...
type	<...> ...
typetag	<...> ...

exists, it is never called. Instead, to load objects from a binary stream, use

read_obj()

Returns	<...> ...

to grab the next object. Check the type of the object returned to decide what is to be done with the object.

dp_obj(*ptr*)

Returns	<...> ...
ptr	<...> ...

dumps objects in a binary stream to standard output.

The bounding box operation

bb_obj(*bbox*, *ptr*)

Returns	<...> ...
bbox	<...> ...
ptr	<...> ...

is also never called directly. Instead, use

bbox_val(*obj*, *minmax*, *coord*)

Returns	<...> ...
obj	<...> ...
minmax	<...> ...
coord	<...> ...

to access a particular value in the bounding box. This routine checks that a bounding box has not already been computed, and calls **bb_obj** only if necessary.

The raster rendering program uses

rdisp_obj(*object*, *ptr*)

Returns	<...> ...
object	<...> ...
ptr	<...> ...

p_rem_obj(*ptr*, *yval*)

Returns	<...> ...

ptr <...> ...
yval <...> ...

The **view** program uses

ldisp_obj(*ptr*)

Returns <...> ...
ptr <...> ...

Debugging, like printing, uses an indentation level:

dbg_obj(*ptr*, *lvl*)

Returns <...> ...
ptr <...> ...
lvl <...> ...

These routines facilitate the handling of lists of objects. They are especially useful for attributes lists attached to objects.

cp_obj_list(*old_list*)

Returns <...> ...
old_list <...> ...

dbg_obj_list(*obj_list*, *lvl*)

Returns <...> ...
obj_list <...> ...
lvl <...> ...

dp_obj_list(*obj_list*)

Returns <...> ...
obj_list <...> ...

ld_obj_list(*list_type*)

Returns <...> ...
list_type <...> ...

fr_obj_list(*obj_list*)

Returns <...> ...
obj_list <...> ...

pr_obj_list(*obj_list*, *level*)

Returns <...> ...
obj_list <...> ...
level <...> ...

Ld_obj_list expects all of the objects in the list to be of the given type.

These routines are needed to support the protocol operations for some objects. They are essentially the protocol operations for some internal structures which are not tagged objects. These internal structures are in some sense the primitives which make up the tagged objects.

A point can be printed for debugging with an optional name field using

dbg_point(*p*, *name*)

Returns	<...> ...
p	<...> ...
name	<...> ...

or put out in Alpha_1 text file format using

pr_point(*p*)

Returns	<...> ...
p	<...> ...

Curve and surface support includes:

cp_curve(*crv*)

Returns	<...> ...
crv	<...> ...

fr_curve(*crv*)

Returns	<...> ...
crv	<...> ...

dp_curve(*crv*)

Returns	<...> ...
crv	<...> ...

ld_curve(*crv*)

Returns	<...> ...
crv	<...> ...

pr_curve(*crv*, *level*)

Returns	<...> ...
crv	<...> ...
level	<...> ...

cp_srf(*srf*)

Returns	<...> ...
srf	<...> ...

fr_srf(*srf*)

Returns	<...> ...
srf	<...> ...

dp_srf(*srf*)

Returns	<...> ...
srf	<...> ...

ld_srf(*srf*)

Returns	<...> ...
srf	<...> ...

pr_srf(*srf*, *level*)

Returns <...> ...
srf <...> ...
level <...> ...

pr_knot_vect(*kv*, *level*)

Returns <...> ...
kv <...> ...
level <...> ...

pr_mesh(*mesh*, *geom_coords*, *level*)

Returns <...> ...
mesh <...> ...
geom_coords
<...> ...
level <...> ...

6.8 Attribute Package

The attribute package provides a set of routines for handling attributes of objects. These routines are similar to those provided in Rlisp for handling property lists.

There is no particular attribute data structure. Attributes of Alpha_1 objects may be almost any "official" object in the system, and any object in the system may have an attribute field. See chapter 5 [Object Management in C], page 61 for information on what Alpha_1 objects actually are. An object may be an attribute only if it has a **LINKS** field (see section 5.3.1 [Str Files], page 67). The routines and macros for managing attributes are described briefly below.

These are the top level macros, which will generally be the only ones that users of the package will need (unless they are using the user defined attributes described later).

To set an attribute of an object, use

SET_ATTR(*attr_type*, *obj*, *attr*)

Returns <...> ...
attr_type <...> ...
obj <...> ...
attr <...> ...

The attribute is returned.

GET_ATTR(*attr_type*, *tag*, *obj*)

Returns <...> ...
attr_type <...> ...
tag <...> ...
obj <...> ...

retrieves an attribute of the object with the given type tag.

REM_ATTR(*tag*, *obj*)

Returns	<...> ...
tag	<...> ...
obj	<...> ...

removes an attribute of the object with the given type tag.

FREE_ATTR(*obj*)

Returns	<...> ...
obj	<...> ...

frees all the attributes of an object. This is equivalent to removing them all one by one.

User defined attributes may be used when an attribute is desired which is not already an official object. User defined attributes are for simple tagging of objects with string, integer, or float values for internal use in a program or communication between cooperating programs. The following functions are provided for using these three types of attributes.

The retrieval routines are:

get_str_attr(*attr_nm*, *obj*)

Returns	<...> ...
attr_nm	<...> ...
obj	<...> ...

get_fl_attr(*attr_nm*, *obj*)

Returns	<...> ...
attr_nm	<...> ...
obj	<...> ...

get_int_attr(*attr_nm*, *obj*)

Returns	<...> ...
attr_nm	<...> ...
obj	<...> ...

To set the attributes:

set_str_attr(*obj*, *attr_nm*, *str_val*)

Returns	<...> ...
obj	<...> ...
attr_nm	<...> ...
str_val	<...> ...

set_fl_attr(*obj*, *attr_nm*, *float_val*)

Returns	<...> ...
obj	<...> ...
attr_nm	<...> ...
float_val	<...> ...

set_int_attr(*obj*, *attr_nm*, *integer_val*)

Returns	<...> ...
obj	<...> ...

attr_nm <...> ...
integer_val
<...> ...

The removal routines are:

rem_str_attr(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

rem_fl_attr(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

rem_int_attr(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

A set of predicates is also provided for finding out whether a particular attribute exists for an object.

attr_p(*attr_tag*, *obj*)

Returns <...> ...
attr_tag <...> ...
obj <...> ...

str_attr_p(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

fl_attr_p(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

int_attr_p(*attr_name*, *obj*)

Returns <...> ...
attr_name <...> ...
obj <...> ...

6.9 Matrix Package

The matrix package defines routines for handling dynamically allocated arrays. For dynamically allocated arrays of more than one dimension, the C language will not handle the indexing automatically. This package provides data structures and routines to handle these arrays with less effort.

They are used extensively by the B-spline curve and surface structures. Currently, the arrays have a maximum of three dimensions, and the elements of the matrix must be floats.

Several datatypes are provided:

matrix Basic matrix structure. Stores sizes of each dimension, strides (for indexing), and the values.

mat1 One-dimensional matrix type.

mat2 Two-dimensional matrix type.

mat3 Three-dimensional matrix type.

vector Same as mat1.

array Same as mat2.

To allocate a vector or matrix, use:

new_vector(*size*)

Returns <...> ...
size <...> ...

and

new_array(*row_size*, *column_size*)

Returns <...> ...
row_size <...> ...
column_size <...> ...

In code dealing with geometric entities, it is often useful to use the last dimension of an array or vector for the coordinates of points.

So other datatypes include:

pt_vector Same as **mat2** (with second index for coordinates).

Hpt_vector Same as **mat2** (with second index for coordinates).

pt_array Same as **mat3** (with third index for coordinates).

Hpt_array Same as **mat3** (with third index for coordinates).

The routines for allocating them are:

new_pt_vector(*size*)

Returns <...> ...
size <...> ...

new_Hpt_vector(*size*)

Returns <...> ...
size <...> ...

new_pt_array(*row_size*, *column_size*)

Returns <...> ...
row_size <...> ...

column_size
<...> ...

new_Hpt_array(*row_size*, *column_size*)

Returns <...> ...
row_size <...> ...
column_size
<...> ...

The routine

pt_size(*mat*)

Returns <...> ...
mat <...> ...

will tell what the size of the points in the matrix are (3 or 4).

Index functions provide random access into dynamically allocated matrices and vectors.

mat_index(*mat*, *dimension*, *i*, *j*, *k*)

Returns <...> ...
mat <...> ...
dimension <...> ...
i <...> ...
j <...> ...
k <...> ...

vec_index(*vec*, *i*)

Returns <...> ...
vec <...> ...
i <...> ...

The "next" macros facilitate sequential access through arrays or vectors.

To increment a pointer through the elements of a vector, use

VEC_NEXT(*element_ptr*, *vec*)

Returns <...> ...
element_ptr
<...> ...
vec <...> ...

Similar operations for matrices are

NEXT_ROW(*element_ptr*, *mat*)

Returns <...> ...
element_ptr
<...> ...
mat <...> ...

NEXT_COL(*element_ptr*, *mat*)

Returns <...> ...

element_ptr
<...> ...
mat <...> ...

NEXT_PLANE(*element_ptr*, *mat*)

Returns <...> ...
element_ptr
<...> ...
mat <...> ...

MAT_NEXT(*dimension*, *element_ptr*, *mat*)

Returns <...> ...
dimension <...> ...
element_ptr
<...> ...
mat <...> ...

Two defined constants are generally used as flags to specify operations on meshes which are in the row or column direction:

ROW The first index of an array.
COL The second index of an array.

The datatype

array_dir

can have **ROW** or **COL** as its value, and

otherdir(*dir*)

Returns <...> ...
dir <...> ...

returns the other value.

Various macros exist for asking a matrix about its dimensions:

mat_size(*mat*, *dim*)

Returns <...> ...
mat <...> ...
dim <...> ...

vec_size(*vec*)

Returns <...> ...
vec <...> ...

row_size(*mat*)

Returns <...> ...
mat <...> ...

col_size(*mat*)

Returns <...> ...

mat <...> ...

depth_size(*mat*)

Returns <...> ...
mat <...> ...

The **sub_vector** routine,

sub_vector(*vec*, *begin_posn*, *length*)

Returns <...> ...
vec <...> ...
begin_posn
<...> ...
length <...> ...

extracts a subsequence of a vector.

Multiplication of two matrices is accomplished using

mat_mult(*mat1*, *mat2*)

Returns <...> ...
mat1 <...> ...
mat2 <...> ...

if they are conformable.

6.10 Mesh Package

The mesh package defines the structures and access operations for the basic components of B-spline curves and surfaces: order, knot vectors, and control meshes. Accessing the mesh package automatically includes the matrix package.

The following definitions and routines handle knot vectors for both curves and surfaces. Knot vectors are ordered sets of floating values constrained to be in non-descending numerical order. The datatypes are

knot
knot_vector

A knot vector is created with

new_knot_vector(*size*)

Returns <...> ...
size <...> ...

Two knot vectors can be merged into one using

merge_kv(*new_knots*, *old_kv*)

Returns <...> ...
new_knots <...> ...
old_kv <...> ...

The datatypes

ctl_polygon
Hctl_polygon
ctl_mesh
Hctl_mesh

hold the B-spline control points (or **Hpoints**) for curves and surfaces respectively.

The allocation routines are

new_c_polygon(*size*)

Returns <...> ...
size <...> ...

new_Hc_polygon(*size*)

Returns <...> ...
size <...> ...

new_c_mesh(*row_size*, *col_size*)

Returns <...> ...
row_size <...> ...
col_size <...> ...

new_Hc_mesh(*row_size*, *col_size*)

Returns <...> ...
row_size <...> ...
col_size <...> ...

The routines

c_poly_size(*c_poly*)

Returns <...> ...
c_poly <...> ...

c_mesh_size(*c_mesh*, *dir*)

Returns <...> ...
c_mesh <...> ...
dir <...> ...

are used to access the size information.

A

curve

consists of its order, a knot vector, and a control polygon. It is created with

new_curve(*order*, *knots*, *ctl_poly*, *ctl_pt_type*)

Returns <...> ...
order <...> ...
knots <...> ...
ctl_poly <...> ...
ctl_pt_type
<...> ...

A

surface

consists of two orders (one for each parametric direction), two corresponding knot vectors, and a control mesh. Note that most programs will want to deal with surface objects (srf_obj) rather than with surfaces.

new_surface(*u_order*,*v_order*, *u_knots*,*v_knots*, *mesh*, *ctl_pt_type*)

Returns <...> ...
u_order <...> ...
v_order <...> ...
u_knots <...> ...
v_knots <...> ...
mesh <...> ...
ctl_pt_type
<...> ...

To insert the control polygon of a curve as the row or column of a control mesh, the routine

put_crv(*ctl_poly*, *mesh*, *indx*, *dir*)

Returns <...> ...
ctl_poly <...> ...
mesh <...> ...
indx <...> ...
dir <...> ...

is provided. No size checking is done by the routine. The inverse routine,

grab_crv(*srf*, *indx*, *dir*)

Returns <...> ...
srf <...> ...
indx <...> ...
dir <...> ...

is used to extract a row or column of the control mesh as a curve.

6.11 Refinement Package

The refinement package provides functions used to refine B-spline curves and surfaces. Refinement in this sense means adding knots (and thus control points) to the curve or surface, in such a way that the augmented knots and control points define exactly the same curve or surface as originally. Accessing the refinement package automatically includes the mesh and matrix packages.

The high level routines (normally, only these will be directly called by programs using the package) are

c_refine(*old_crv*, *nw_knots*, *do_merge*)

Returns <...> ...
old_crv <...> ...
nw_knots <...> ...

do_merge <...> ...

and

s_refine(*old_srf*, *dir*, *nw_knots*, *do_merge*)

Returns <...> ...
old_srf <...> ...
dir <...> ...
nw_knots <...> ...
do_merge <...> ...

The low level datatype

alpha_mat

is used in the refinement routines.

The low-level refinement operations include

new_alpha(*order*, *nrows*, *ncols*)

Returns <...> ...
order <...> ...
nrows <...> ...
ncols <...> ...

free_alpha(*alpha*)

Returns <...> ...
alpha <...> ...

calc_alpha(*order*, *tau*, *t*, *work_area*)

Returns <...> ...
order <...> ...
tau <...> ...
t <...> ...
work_area <...> ...

alpha_mul(*old_mesh*, *dir*, *alpha*, *beg_pos*, *length*)

Returns <...> ...
old_mesh <...> ...
dir <...> ...
alpha <...> ...
beg_pos <...> ...
length <...> ...

An alpha matrix created by **new_alpha** (or **calc_alpha**) should be freed with **free_alpha** (and **not** free).

6.12 Spline Package

The **d_spline** package provides some high level definitions and routines for handling B-spline curves and surfaces. Accessing the **d_spline** package automatically includes the refine, mesh, and matrix

packages.

A few miscellaneous defined constants are used to make the subdivision code clearer.

BIGVAL A very large floating point number.
NO Equivalent to FALSE.
YES Equivalent to TRUE.
STRAIGHTERR
Tolerance for straightness testing, currently 1.05 value.

These constants are used to record how far the flatness has proceeded.

TEST_FLAT
Unknown value for flatness.
FLAT Value signifying surface is flat.
VERT Value indicating flatness in vertical direction (same as ROW).
HORIZ Value indicating flatness in horizontal direction (same as COL).

The datatype

split_code

may take on those flatness values.

A subdivided surface is stored in an intermediate structure

srf_halves

consisting of the two subdivided halves until attributes and other relevant information can be propagated from the parent surface. The defined constants

FIRST
SECOND

are used to access the two surfaces in this structure.

Edge straightness is required before flatness can be achieved, and the defined constants

TOP Value signifying top curve of mesh is straight.
BOTTOM Value signifying bottom curve of mesh is straight.
LEFT Value signifying left curve of mesh is straight.
RIGHT Value signifying right curve of mesh is straight.

are values that may be taken on by a variable of type *edge_code*.

These are the actual subdivision routines. **Spl_eval** and **spl_init** are the usual ones for users of the subdivision package to call, but it is sometimes useful to call **srf_subdiv** and **sep_halves** directly.

The top level subdivision routine is

spl_eval(*srf*)

Returns <...> ...
srf <...> ...

returns either a subdivided srf_obj or four polygons. The srf_objs are set up for subdivision using

spl_init(*srf*, *tran*)

Returns <...> ...

srf <...> ...
tran <...> ...

Intermediate level routines include

srf_divide(*srf*, *dir*)

Returns <...> ...
srf <...> ...
dir <...> ...

which subdivides a surface at the parametric center of the knot vector, and

srf_cut(*srf*, *dir*, *mid_index*, *multiplicity*, *knot_value*)

Returns <...> ...
srf <...> ...
dir <...> ...
mid_index <...> ...
multiplicity <...> ...
knot_value <...> ...

which is like **srf_divide**, except that you choose the subdivision point.

At the bottom of all those are eventually calls to

srf_subdiv(*srf*, *dir*, *mid_index*, *split_mult*, *knot_val*)

Returns <...> ...
srf <...> ...
dir <...> ...
mid_index <...> ...
split_mult <...> ...
knot_val <...> ...

which does the actual subdivision, and

sep_halves(*halves*, *srf*, *dir*)

Returns <...> ...
halves <...> ...
srf <...> ...
dir <...> ...

which separates the two surface halves into distinct srf_objects, propagating some information from the parent surface.

After subdivision, the finishing routines are

srf_output(*srf*)

Returns <...> ...
srf <...> ...

which subdivides the surface at any remaining multiple knots and returns a set of approximating polygons. This routine uses

mult_knots(*srf*, *dir*, *index*, *mult*, *value*)

Returns	<...> ...
srf	<...> ...
dir	<...> ...
index	<...> ...
mult	<...> ...
value	<...> ...

to check a surface for multiple knots, and return the index of the first occurrence. Finally,

srf_to_poly(*srf*)

Returns	<...> ...
srf	<...> ...

divides a (flat) surface up into polygons.

Information for shaded rendering comes from

srf_corners(*srf*, *parent*, *dir*, *half*)

Returns	<...> ...
srf	<...> ...
parent	<...> ...
dir	<...> ...
half	<...> ...

srf_normals(*normals*, *mesh*)

Returns	<...> ...
normals	<...> ...
mesh	<...> ...

The routines that do the actual flatness and straightness testing are

test_flat(*srf*)

Returns	<...> ...
srf	<...> ...

which determines whether a surface is flat, and returns a split_code,

test_strght(*srf*)

Returns	<...> ...
srf	<...> ...

which checks that the edges of a surface are straight, and

crv_strght(*srf*)

Returns	<...> ...
srf	<...> ...

which checks a curve in a surface for straightness.

6.13 Menu Package

(Need a volunteer to write this.)

6.14 Symbol Table Package

A simple symbol table package may be accessed by including "symtab.h" in the source file. The symbol table package defines a structure called an *id* which contains the symbol name and a value which may be associated with it (usually a pointer to some other object). The package also defines a structure called a hash_table, and routines for creating and accessing this structure.

Creating an id and putting it in the hash table is accomplised using

new_symbol(*sym_name*, *table*)

Returns <...> ...
sym_name <...> ...
table <...> ...

To retrieve an id previously created use

find_symbol(*sym_name*, *table*)

Returns <...> ...
sym_name <...> ...
table <...> ...

Creating a new hash table requires specification of the number of entries:

new_hash_table(*n_entries*)

Returns <...> ...
n_entries <...> ...

A vector of already initialized id's can be linked into the hash table using:

ld_table(*id_vec*, *table*)

Returns <...> ...
id_vec <...> ...
table <...> ...

Two macros for initializing symbol vectors are provided:

VAR_SYM(*name_string*, *var_name*)

Returns <...> ...
name_string
<...> ...
var_name <...> ...

FN_SYM(*name_string*, *fn_name*)

Returns <...> ...
name_string
<...> ...
fn_name <...> ...

7. Objects in Rlisp

7.1 PSL Object Overview

The lisp sources which make up the **shape_edit** program rely heavily on an "object-oriented" style of programming. Previous chapters have described the C "object" and "protocol" concepts used in Alpha_1 system utilities to achieve an object-oriented approach. A similar approach with a different implementation is used in the Alpha_1 lisp code.

This document does not attempt any exhaustive explanation of object-oriented programming, as many articles are available on the subject. PSL, the lisp system on which **shape_edit** is built, has an object package available. A portable interface layer has been implemented so that the implementation details of the particular object package which we use are hidden from the majority of the routines in the Alpha_1 system.

The basic idea of object-oriented programming is to represent data by a collection of objects and to manipulate the data by performing operations on the objects. An object consists of a datastructure containing its state, and a set of procedures for interpreting the state and carrying out actions or operations on the object. Individual datastructures of one type are called "instances" of the object type. The internal datastructure used to represent the state of the object is private. The **only** way to access the state of an object is using the operations which are defined for the object. These operations are called "methods" in the **shape_edit** environment.

7.2 DefObject

A description of the objects package which is available in PSL is described in the PSL Manual for Version 3.4. However, Alpha_1 applications should only use the facilities in the "DefObject" package described in this section. Most of the time, applications will want to use **defModelObject** (described in the next section) for objects which could be part of a geometric model, rather than the "bare" defObject described here.

The simplest form for defining an object type is:

```
defObject( ObjName,
           Slot1,
           Slot2, ... );
```

The first argument gives a name for the object type, and remaining arguments give the names of slots or elements in the object datastructure. For example, a point might be defined as:

```
defObject( Point,
           X, Y, Z );
```

There are a number of options which may be associated with the object type. These include:

!:Creator The creator function for instances of the object is given the specified name. If the name is not given, the creator function is called **create***ObjName*, (where *ObjName* is the first argument to **defObject**). If this option is not given, no creator function is defined.

!:Constructor
The constructor function for instances of the object is given the specified name. If this option is not given, the constructor function is called

make*ObjName*. A constructor function is always defined unless "!:Constructor Nil" is specified.

!:Predicate
: The predicate function for instances of the object is given the specified name. If this option is not specified or no name is given, the predicate function is called *objName*P. A predicate function is always defined, unless "!:Predicate Nil" is specified.

!:InheritFrom
: The object inherits from the other object specified.

!:AllSettable
: All slots of the object may be assigned (implies **AllGettable** and **AllInitible**). This is the default.

!:AllGettable
: All slots may be examined.

!:AllInitable
: All slots may be initialized on creation of an instance.

!:AllPrivate
: All slots are private (may not be set or accessed except by methods of the object).

These options are specified as "arguments" (like a function call) of the object name if any are given; they are enclosed in parentheses and separated by commas. So, if we wished to create points using a function called "point", and to make the point positions unable to be changed after they are defined, we could change our previous example to read:

```
defObject( Point( !:AllGettable, !:AllInitable, !:Creator point ),
           X, Y, Z );
```

There are also options which may be associated with the individual slots of the object. These are:

!:InitExpr You may specify a default initialization expression to be used to fill in slots if they are not specified on creation of an instance. A default initialization slot option must be the first slot option specified.

!:Type To specify the type of the slot. Currently this is really just a comment which gives some notion of the intended use of the slot. Alpha_1 sources, as a documentation mechanism, should **always** use the type option for all slots.

!:Settable The slot may be assigned values. Implies **Gettable** and **Initable**, and is the default.

!:Gettable The slot may not be assigned, but can be examined.

!:Initable The slot may be initialized on creation.

!:Private The slot is private, accessible only through methods of the object.

!:NotCopied
: The slot is not copied by copying methods.

Options for slots are specified just like the object options above: the options follow the slot name in parentheses, with commas as separators.

Again, our point example would be better written as:

```
defObject( Point( !:AllGettable, !:AllInitable, !:Creator point ),
```

```
X( 0.0, !:type Float ),
Y( 0.0, !:type Float ),
Z( 0.0, !:type Float ) );
```

where the 0.0 initialization values guarantee that the slots will have a value even if all three coordinates are not specified when the instance is created.

The **defObject** package provides a routine for determining if a given method is defined for a given object.

handlesP(*Object*, *MethodId*)

Returns <...> ...
Object <...> An instance of an object.
MethodId <...> The name of the method (will usually need to be quoted).

You will usually need to quote the method name. The return value is Nil if the method is not defined. Otherwise it is the name of the procedure which implements the method for that object type.

The function **object!-type** will return the type of the given object.

object!-type(*object*)

Returns <...> ...
object <...> ...

The **defObject** package also provides a convenient way of accessing slots in an object using the "->" construct, as in structure reference through a pointer in C. Assuming that the slot is gettable, it may be accessed as

```
instance->slotname
```

This expression may also appear on the left side of an assignment if the slot is settable.

Procedures Defined by DefObject

A number of procedures are defined automatically when you define an object using **defObject**.

There is always a predicate function, which tells you whether an instance is a particular kind of object. The name of the function depends on the !:predicate option described above, but is usually just the object name with suffix "P" (e.g., **surfaceP**).

A constructor function is also automatically defined. Its name depends on the !:constructor option. The constructor function takes arguments for slot assignments in keyword-value pairs, where the keyword is the name of the slot. For the point example above, we could create an instance with given X and Z coordinates while letting the Y coordinate assume its default initialization as follows:

```
makePoint( z 1.0, x 2.0 );
```

If the !:creator option was given, a creation function which requires all slot values to be specified in a fixed order is defined. The name of the function depends on whether one was specified in the !:creator option. For the point example, an instance could be created with:

```
p := point( 1.0, 2.0, 3.0 );
```

In addition to these functions, several methods are defined for the object. One is **describe** which prints information about an instance and its state, as in:

describe(*p*)

Returns <...> ...

```
p           <...> ...
```

A **channelPrin** (standard printing) method is also defined for each object. The default **channelPrin** methods are usually unsatisfactory, and you may want to override them with new ones. Unfortunately, the **channelPrin** methods are difficult to define well, and are best done by finding an existing one for a similar object in the system and modifying the code appropriately.

Depending on options, two methods may also be defined for each slot in the object: one for setting values (if the slot is settable) and one for examining values (if the slot is gettable). Both of these are accessible through a function named by the slot. So

```
y( p );
```

will yield the y coordinate of point p in our example. If the slots of point were settable, we could set the y coordinate using

```
y( p ) := 4.0;
```

The "->" symbol discussed above actually translates to calls on the slot function. The constructions

```
p->y
```

and

```
p->y := 4.0;
```

are equivalent to the two shown above, and are preferred for clarity, as well as some subtle syntactic precedence reasons.

You need not worry about whether slot names conflict with slot names in other objects. Even though the same function is used from the top level, the underlying procedures are really methods which are correctly invoked for the specific object you provide.

7.3 DefModelObject

A "model object" is just like other objects, except that it has some extra specialized slots which are used by the modeling system. The most important features of a model object are that it can be displayable (although it need not be) and "dependency propagation" occurs during construction and manipulation of the model object.

Dependency propagation will require a chapter of its own to explain fully. The basic idea is that an object "remembers" other objects which were used in its construction. If any of those objects are changed, the dependent object will be updated automatically (including its graphical representation on a display if necessary).

If you need either display or dependency capabilities, you should use **defModelObject**, rather than **defObject** to define your object type. Consult an Alpha_1 staff member if you need help determining whether an object type is a model object type or not. (You may also be able to use the higher level **defParamType** described in the User's Manual).

The syntax for **defModelObject** is just like that for **defObject** described above. Your object type will automatically inherit from an object type called **ModelObj** (which you should never see). All other options are as in **defObject**.

The object type will contain a number of fields which are used to implement the display methods (if defined) and the dependency propagation. These are not relevant to this discussion, but two other extra fields are useful to know about.

Each model object will have a "name" slot which allows a string name to be associated with an object. Often, this slot will be filled in when you assign the object to an Rlisp variable. So if you create a point and assign it to the variable "point1", as in

```
point1 := pt( 1.0, 2.0, 3.0 );
```

the name slot will contain "point1". If an object already has a name when the assignment is done, it will keep its original name.

Each model object will also have an aList slot for storing attributes of the object. These might include color or resolution for rendering. The functions for manipulating attribute lists are described in the section on attributes in the Alpha_1 User's Manual.

The **modelObjectP** function will return T if the instance given as argument is a model object.

A copying method is defined for each model object, which is accessible through the **totalCopyGeom** function. It copies all the fields of an object except those which may have been tagged with a !:NotCopied option. A printing method (channelPrin) is also defined for the object. This is the routine which will be invoked by the PSL whenever it needs to print your object.

7.4 Defining Methods

Once an object is defined using **defObject** or **defModelObject**, it will usually be necessary to define some methods for the object which are not automatically generated by the system. A method is similar to any other procedure. It uses a special "method" keyword, contains the name of the object type for the method as well as the method name. The object to which the operation is applied is treated specially. When invoking the method procedure, the first argument is always the object, but this argument is **not** specified when defining the method. The slot accessors of the object are treated as if they were local variables within the method procedure. If the object must be referenced (for example to apply another method to it), its name within the method procedure is "Self".

As an example, suppose we wanted to write a "length" method for the point object example we have been using. The length method computes the length of a vector from the origin to the point instance:

```
method procedure point length()
        sqrt( (X * X) + (Y * Y) + (Z * Z) );
```

The "method procedure" keywords are followed by the object name and the method name. This example has no formal parameters, but they would be specified as usual for procedures. The slots are just accessed by giving their names. Saying "Self->X" rather than "X" is equivalent but a little less efficient due to the implementation of the objects package.

Contrary to the appearance in the method definition, this is not a function with no arguments; you must of course provide the object the method is to be applied to, as in:

```
length( p );
```

We almost always invoke methods using a "send macro" which has the same name as the method, rather than using the "send" routine provided by the objects package directly. So we would say

```
mapObj( TestObject, Trans );
```

rather than

```
send( TestObject, 'mapObj, Trans );
```

which is the access provided by the objects package. The send macro is defined automatically when the method is defined, but for other code which may also call the send macro you must declare the method in an appropriate rlisp definitions file using

declareMethod(*MethodName*)

Returns <...> ...
MethodName
<...> ...

or

declareMethods(*MethodName1*, *MethodName2*, ...)

Returns <...> ...
MethodName1
<...> ...
MethodName2
<...> ...

A few words about when to use methods rather than simple procedures are in order. Operations which are "generic", rather than specific are the ones that should be methods. One hint might be that you can imagine using the same operation name on other objects. Generic operations that you would want to apply to lots of different objects could include "copy", "display", "writeToFile", "boundingBox" or "prettyPrint". An operation that really only makes sense when applied to one kind of object, like **evaluateCurve** would be a simple procedure rather than a method.

7.5 Standard Methods used in Alpha_1

This section discusses some of the methods which are provided for a number of Alpha_1 geometric objects. If you create a new object, you will probably need to define some of these methods in order for the object to be fully integrated into the **shape_edit** utilities.

In all cases, you should examine the methods which are already defined for other objects to get an idea of an appropriate implementation for the new object. In many cases, the method for a new object will be a simple "destructuring" of the object and applying the method to the subpieces. For example, a shell object contains a list of surfaces, and the display method for shells simply forms that list of surfaces into a group and invokes the group display method.

One of the first methods you will want to consider for a new object is the **channelPrin** method. A default is provided when the object is defined (with defObject, then **defModelObject** makes another attempt), but Alpha_1 objects sometimes override even that with one that formats the data as nicely as possible. As mentioned earlier, **channelPrin** methods are somewhat tricky, so be sure to model yours on an existing one.

Another printing methods may be useful, if there is or will be an Alpha_1 text file format for the object. This is **dumpA1**, which dumps the object to a text file Creating a text file format for new objects involves non-trivial modifications to the C parser used for all text files. However, defining a **dumpA1** method which will output the object (ignoring for the moment whether any programs can actually read it) is quite simple. You can debug the **dumpA1** methods easily by calling **dumpA1File** with Nil for the file argument. This causes the output to be printed on your screen.

If your object contains embedded point or vector data, you will probably want to provide all of the following methods: **commonType**, **ptCoerce**, **ptVecCoerce**, and **mapObj**. The **commonType** method applied to an object insures that all the points in the object are of the same type. The

ptCoerce method coerces all the points in the object to a given type, and **ptVecCoerce** applies coercions to both points and vectors. The **mapObj** method is the basic graphical transformation utility. It applies a given transformation matrix to all points and vectors in the object.

Many objects will need an appropriate graphical representation for visual feedback, and so require a **dlsObj** method. If the graphical representation is composed of existing objects which can be displayed, the **dlsObj** method is a simple destructuring operation as described for the shell example earlier in this section. If there is some reason that you think this approach is inappropriate for a new object, you should discuss it with a member of the Alpha_1 staff as modifications to the display manager specification file (and hence all current device support) will be required.

If the new object has an implicit orientation (most of our geometric objects — lines, planes, surfaces — have orientation), then you will want to provide a **reverseOrient** method which allows the orientation to be reversed using the **reverseObj** function without regenerating all the geometry.

7.6 Procedures Using Model Objects

We have mentioned only briefly the dependency propagation utilities which are provided for all objects which are defined using **defModelObject**. In order for the dependency propagation to work correctly, there are some special declarations and keywords which are required when writing procedures involving model objects.

A procedure which involves model objects will be classified as one of:

- constructor procedure
- extractor procedure
- modification procedure
- internal geometry procedure

The first step in the classification is to decide if the procedure is something that a user is likely to call during model construction, or whether it is purely an internal function which is only used by other system functions. For example, the "bend" operation uses a helper function which is called "bend2". Since bend2 should never be called by a user, but only by the bend function, bend2 is declared as an internal geometry procedure.

If the procedure is not internal, then consider how it operates on the objects. A procedure which constructs and returns a model object is a constructor procedure. A procedure which extracts information (but not object-valued information) from a model object is an extractor procedure. Constructor takes precedence over extractor, so if a procedure extracts a point from another model object, it will be declared a constructor since a point is also a model object. A procedure which extracts the radius parameter from a sphere object, though, is an extractor since a radius is not a model object. Finally, some functions just modify an object. An example is **mkAdjacent** which adds an adjacency attribute to each of two surfaces. It modifies the attribute list, but does not construct new geometry. Currently there is no special declaration for modifiers: modification procedures should be declared as extractor procedures.

If the procedure is a method for an object, then the same rules for classifying the procedure apply, but the keywords have "Method" appended to them: InternalGeometryMethod Procedure, ConstructorMethod Procedure, and ExtractorMethod Procedure.

Any procedures which are defined as constructor or extractor must also be declared in an rlisp definitions file (just like simple methods). The **declareConstructors** and **declareExtractors** statements are provided for this purpose. These declarations insure that code which calls those routines will be compiled correctly. Both of these take precedence over **declareMethod**, so a **constructorMethod**

procedure should be listed in a **declareConstructors** statement, rather than a **declareMethods** statement.

> **declareConstructors**(...)
>
> *Returns* **<...>** ...

and

> **declareExtractors**(...)
>
> *Returns* **<...>** ...
>
> **declareExtractor**(...)
>
> *Returns* **<...>** ...

There is a reason there are two forms of **declareExtractors**, but only one of **declareConstructors**. The logic which links modeling statements together with a network of model objects must be able to know when model objects are being constructed and referenced. **DeclareConstructors** takes a series of constructor names as arguments, all of which are declared to return model objects which will be entered into the model data structures as they are constructed. **DeclareExtractors** (the plural form) similarly takes a list of extractor names and declares that the first (or only) argument of each extractor will be a model object of some type. **DeclareExtractor** (the singular form) takes an extractor name as its first argument, and the rest of the arguments are argument numbers telling which of the arguments will be model object valued. (The first argument in calls on the extractor is argument number 1.)

To summarize, here are the rules for classifying a procedure that deals with model objects:

InternalGeometryMethod Procedure:
InternalGeometry Procedure:
: Although it may have model object arguments or return value, this procedure is never intended to be called from the user level during model construction.

ConstructorMethod Procedure:
Constructor Procedure:
: This procedure creates and/or returns a model object. It may or may not have model objects as arguments. It does not always have to return a model object — if it ever does, it is a constructor. For example, crvEval is a constructor because it can return points or vectors, even though sometimes it returns scalars and lisp vectors. Be sure that the definitions file containing the declaration is loaded (with "bothTimes load") in any files that have procedures which call your procedure.

ExtractorMethod Procedure:
Extractor Procedure:
: This procedure has one or more model objects as arguments, and returns a value which is **never** a model object.

ModificationMethod Procedure:
Modification Procedure:
: This procedure has one or more model objects as arguments, and modifies the objects, but does not return a value. Modification procedures are currently declared as extractors.

The following table lists the procedure types and the required declaration statements as a summary.

Procedure Type
Declaration

constructor
declareConstructors

extractor **declareExtractors**

internalGeometry
none

method **declareMethods**

constructorMethod
declareConstructors

extractorMethod
declareExtractors

internalGeometryMethod
none

It is worthwhile to think carefully about these declarations. Unfortunately, it is very easy to discover that things are not getting declared and compiled correctly, and very difficult to decide exactly **what** isn't getting declared. The most common way that errors are detected is that some routine dies while attempting to process (or returns) an entity which looks like:

(!&someFunctionName ...)

This means that some other routine which **calls** the named function was compiled without appropriate function declarations loaded. These errors show up only in compiled code — so interpretively tracking through all the code will not solve the problem.

8. Rlisp Packages

8.1 Rlisp Package Introduction

Rlisp packages, like C packages, are groups of data structure definitions, macros, global variables, defined constants, procedure declarations, and procedures. An Rlisp package consists of one or more "definition files" which are analogous to the ".h" files in the C packages, and a group of files which define the associated functions of the package.

Rlisp packages may be organized around objects or around a coherent set of functions. For example, the geometry package contains objects which represent simple geometric constructs like lines and arcs. The interpolation package, on the other hand, implements a set of routines which perform various types of spline interpolation.

An Rlisp definition file always is installed on the **shape_edit** include directory $shi. It should usually end in "-df.r", as in "arcs-df.r" and "lines-df.r". A definition file must contain all definitions which need to be known at compile time by the other files in the package. This includes all macros as well as **defObject**, **defModelObject**, **defSymbol**, **declareMethod**, **declareConstructor**, and **declareExtractor** statements. In general, definition files should not contain regular procedures or methods unless they are used at compile time by macros.

All of the files which need the definitions for compilation must use

    bothTimes load defn!-df;

to cause the files to be loaded during compilation. This is exactly analogous to

    #include "defn.h"

in the C code.

Many of the packages which are described in the rest of this section are given only a cursory treatment, listing a few of the functions which they declare. This is because often most of the functions which make up the package are user level functions and are described in full detail in the Alpha_1 User's Manual.

There are a number of files which can be loaded to get collections of small packages loaded in a single step. The most important of these is "geom-df.r" which loads all of the basic objects except splines. The "b-spline-df.r" file loads basic geometry and spline geometry. One of these is usually necessary for files containing subroutines that do any geometric calculations at all. When defining new objects in a "-df.r" file, it will probably be necessary to load either "model-df.r" or "paramtype-df.r", but not any of the geometry packages.

8.2 Miscellaneous Utilities

This section describes a nubmer of little helper functions that may be useful, although they are not properly a "package".

The **defSymbol** macro is used in **shape_edit** to mimic the C "#define" construct. As an example, the order keywords for splines (**LINEAR**, **QUADRATIC**, etc.) are **defSymbol** values.

defSymbol(*Id*, *Val*)

Returns	<Nil> Define a symbol with the given value.
Id	<symbol> The name for the symbol.
Val	<anything> The value for the symbol.

The **idConcat** procedure concatentates two identifiers together to form a new identifier. This is especially useful in code that generates more code.

idConcat(*I1*, *I2*)

Returns	<keyword> Concatenate two keywords to form a longer keyword.
I1, *I2*	<keyword> The two keywords to be concatenated.

Several more list manipulation functions are available.

insertBeforeNth(*ItemToInsert*, *List*, *N*)

Returns	<list> Insert the new item before the Nth one in a list.
ItemToInsert	<anything> The item to insert.
List	<list> The list in which to insert it.
N	<integer> Where to insert it.

insertAfterNth(*ItemToInsert*, *List*, *N*)

Returns	
Returns	<list> Insert the new item after the Nth one in a list.
ItemToInsert	<anything> The item to insert.
List	<list> The list in which to insert it.
N	<integer> Where to insert it.

deleteNth(*List*, *N*)

Returns	<list> Delete the Nth item from a list.
List	<list> The list from which something is to be deleted.
N	<integer> Index of the item to be deleted.

A number of approximate equality functions are provided because exact equality of floating point numbers is rarely meaningful. The **apxEq**, **apxPtEq**, and **apxProjPtEq** functions check approcimate equality of floats, points, and projective points respectively. They use the global variable **ApxEqEpsilon!*** to decide how close things have to be. For **apxPtEq**, two projective points are approximately equal if their Euclidean projections are approximately equal, even if they have very different W coordinates. use **apxProjPtEq** for coordinate-wise equality. The same functions with suffix "Eps" allow the epsilon value to be specified as an argument.

apxEq(*A*, *B*)

Returns	<boolean> Returns a T value if the two are approximately equal.
A, *B*	<number> The values to compare.

apxEqEps(*A*, *B*, *Eps*)

Returns	<boolean> Like **apxEq**, but you get to set the epsilon.
A, *B*	<number> The values to compare.

Eps <float> The epsilon to use for comparison.

apxPtEqEps(*Pt1*, *Pt2*, *Eps*)

Returns <boolean> Coordinate-wise **apxEq** for points, with specified epsilon.
Pt1, Pt2 <point> The points to be compared.
Eps <float> The epsilon to use for comparison.

apxPtEq(*Pt1*, *Pt2*)

Returns <boolean> Coordinate-wise **apxEq** for points.
Pt1 <point> The points to be compared.

apxProjPtEqEps(*Pt1*, *Pt2*, *Eps*)

Returns <boolean> Coordinate-wise **apxEq** for points, but P2 ad P3 points are projected to euclidean space first. Epsilon specified.
Pt1, Pt2 <point> The points to be compared.
Eps <float> The epsilon to use for comparison.

apxProjPtEq(*Pt1*, *Pt2*)

Returns <boolean> Coordinate-wise **apxEq** for points, but P2 ad P3 points are projected to euclidean space first.
Pt1, Pt2 <point> The points to be compared.

Four functions are provided for approximate inequality as wwell. The **apxLessP** (**apxGreaterP**) function corresponds to fuzzy less than (greater than) or equal to. That is, a true value can be returned if the values are within epsilon of each other. The **defLessP** (**defGreaterP**) function returns true only if one value is "definitely" less than (greater than) the other. This is a fuzzy equivalent of strict less than ("<"), while **apxLessP** is a fuzzy equivalent of less-than-or-equal-to ("<=").

apxLessP(*A*, *B*)

Returns <boolean> True value if *A* is approximately less than *B*.
A, B <number> The values to compare.

apxGreaterP(*A*, *B*)

Returns <boolean> True value if *A* is approximately greater than *B*.
A, B <number> The values to compare.

defGreaterP(*A*, *B*)

Returns <boolean> True value if *A* is definitely greater than *B*.
A, B <number> The values to compare.

defLessP(*A*, *B*)

Returns <boolean> True value if *A* is definitely less than *B*.
A, B <number> The values to compare.

8.3 Modeling Package

IOU - Modeling Package (rdf)

8.4 Dependency Propagation

The dependency propagation which occurs automatically for model objects is based on two definition files: "model-df.r" and "construction.r". The macros and Rlisp syntax are in "model-df.r" and some support functions are in "construction.r".

This section describes the top level structures of the model construction and maintenance support, which may become visible to a programmer just below the level where modeling users work.

The propagation of dependency information during a modeling session is dependent on the setting of a global variable, **modeling!***.

on **modeling**

off **modeling**

<boolean> Set flag for maintenance of model connections (default on).

System code (code which is compiled and loaded into **shape_edit**) is by default built with **modeling** turned off. This means that the dependency propagation package generally tracks all user-level constructions, but suppresses the details of constructions within system routines. System routines may occasionally want to use the dependency mechanisms for small sections of code. The **modeling** function turns on modeling for execution of the expression given as an argument. This is used by the animation package, for instance, when it grafts camera and timer objects info the model.

modeling(...)

Returns Whatever the code contained in the arguments returns.

... The code which is to be executed with modeling on.

Basic Concepts

First, we will describe the structure that ties a collection of separate geometric objects together into a model. A model is an acyclic, directed graph of model objects, where each object is linked to the objects that reference it by dependency relations and to objects that it references by prerequisite relations. Dependency and prerequisite relations are inverse; we normally diagram dependency relations as downward arrows between objects and prerequisite relations as upward arrows.

The basic idea of dependency propagation is that an object "remembers" how it was constructed. Its prerequisites are other objects which were referenced in its construction. If any of those objects are changed, the dependent objects will be updated automatically (including their graphical representations on a display if necessary).

Construction expressions are "captured" by constructor macros, and associated with the constructed model objects. Static analysis of the Lisp construction expression trees is done by constructor macros, based on the knowledge stored by **declareConstructor** and **declareExtractor** declarations. A complex construction expression which constructs intermediate model objects is "pruned" into pieces of construction expression corresponding to the various objects.

In a real sense, the **shape_edit** model is a combination of procedures and data structures that is collected incrementally. A model acts like one large data flow procedure whose control structures follow the topology of the references between model objects.

Constructors

When a constructor procedure is defined, it actually defines two Lisp procedures. A little constructor macro is wrapped around an internal constructor function which contains the text of the geometric construction. The name of the constructor function is prefixed with an ampersand ("!&", in Rlisp syntax) to hide it from direct view.

The constructor macro captures and analyzes the argument expressions of a call on the constructor as the macro is expanded. The code returned by the macro is a call on the **addToModel** function, the run-time model object construction function. **AddToModel** constructs an initial object instance, stores the analyzed construction expressions in the modelObj construction slots of the constructed object instance, and links the object into the model using the modelObj dependency slots of the constructed object and its prerequisites.

The body of the construction function is expanded with the **modeling** flag off, so that intermediate objects constructed within a constructor procedure are not linked into the model. A constructor procedure encapsulates a construction and only its returned model object value becomes part of the model. The implication of this is that you can redefine constructor procedures during a modeling session and the changed logic will take effect on model objects constructed using the constructor when they are next changed.

ModelObj Construction Slots

Each geometric object is an instance of a object type defined by **defModelObject**, inheriting slots and methods from the modelObj object type.

ConsFn slots contain the pruned portion of a construction expression that constructed a particular model object. In the general case, it is structured as a

    (function (lambda (args...) body)),

so a model may be compiled properly. (The Lisp lambda s-expression is converted into a code pointer to compiled code by the compiler.) In the special case where the arguments of the lambda would just carry variable values into a call on an internal construction function, the **ConsFn** degenerates to the ampersand-prefixed ID of the construction function. All three possibilities for the value of the **ConsFn** slot (Lisp lambda expression, code pointer, or function ID) may be invoked with a list of run-time arguments by the PSL "apply" function.

Several slots are used to remember information about the construction expression and its analysis for future reference and debugging. The **ConsExpr** slot contains the Lisp expression which was originally given to construct the object and was captured by the constructor macro. The **VarArgs** slot is a list of variable IDs which were referenced in the construction. The **FormalArgs** slot records the lambda expression arguments, which will be a mixture of the variable names and IDs of the form Obj1, Obj2, etc. The **ActualArgs** slot records the pruned subexpressions whose values will be fed to the **ConsFn** to take the place of the **FormalArgs** at run time. For variable references, the formal and actual arguments will both be the variable ID.

ModelObj Dependency Slots

Dependents slots contain a list of other model objects which have referenced an object in their constructions. **Prereqs** slots contain a list of the run-time values of the **ActualArg** expressions. These values may or may not be model objects, since a constructor procedure is not obligated to always return a model object. It might return a scalar number or Nil sometimes, for instance. Also, in cases where it cannot be determined at macro expansion time whether a model object might be present (for example when a variable is referenced) the safe course is taken and the examination of the actual value is deferred until run time.

You can use the **filterModelObjects** function to gather the model objects out of a prerequisites list. The **modelObjectP** predicate returns T if its argument is an instance of an object type defined with **defModelObject**.

The **LinkedToDependents** slot is used to distinguish newly-constructed geometry from "old" geometry which has already been linked into the dependency network. (The need for this is discussed below.)

Constructing Objects and Propagating Changes

When a model object is constructed and added to the model, the function **AddToModel** is invoked with three arguments that will be deposited in the modelObj slots of the newly constructed object instance: the **ConsFn**, the list of **Prereqs** which are its arguments, and a quoted list of construction slot values from the analysis of the construction expression.

To construct a new model object, the **ConsFn** is applied to the **Prereqs** list using the PSL "apply" function. A fresh instance of the proper type results and is linked into the model.

When changes are being propagated, fresh object instances with changed geometry are constructed in exactly the same way as the original object, and the new geometry is copied into the old model objects which were linked into the model. In this way, the object references (tagged Lisp pointers) remain unique "handles" on parts of the model. A function named **propagateChangesFrom** takes a list of model objects which have been changed and propagates the changes through the model.

Model Object Assignment

Model objects, once constructed, have to be assigned somewhere in the Lisp environment in order to be remembered. The RLisp ":=" assignment operator is translated into the modelSetF macro in the modeling environment, rather than its usual translation to the Lisp "setF" macro.

ModelSetF differs from **setF** in that it performs an **updateModelObject** action if the assignment destination already contained a model object. **UpdateModelObject** updates the old model object with the geometry of the new one, just as in change propagation.

We also allow editing the construction expression for a model object assignment and re-entering it into **shape_edit**. This gives **updateModelObject** a somewhat harder job to do. It has to match the existing network of model objects against the newly constructed geometry, deciding which parts match object-for-object and which parts are different. The matching parts are updated in place. The parts which do not match are replaced by unlinking the old geometry from the model and linking the new objects in. Finally, changes are propagated from everything which was changed.

UpdateModelObject considers it to be an error to change the type of an object that has been referenced (linked to dependents in the model.) This is not quite accurate, since there are constructors such as **profile** that take several different types of arguments, but the information to analyze that at run time is currently not available in **shape_edit**. The **forgetName** function removes the binding of the variable IDs you give as its arguments, which helps in some of these cases.

All of the analysis based on constructor and extractor declarations just statically distinguishes portions of model construction statements that could evaluate to model object values from those that couldn't. It doesn't deal very well with cases where the declarations are inadvertently left out or wrong. A true, full type-inference facility in the Lisp system would be far preferable to these ad hoc methods.

8.5 Number Objects

IOU - Number Objects (rdf)

8.6 FloatArray

The floatArray package was developed as part of the effort to begin pushing computationally intensive code from lisp down into C. This requires some data structure compatibilities in order to avoid spending a lot of time formatting arguments at the boundary between the Lisp and C code. In

particular, the most important objects used in the modeler (spline curves and surfaces, knot vectors, transformation matrices) tended to be arrays of floating point numbers. In C, these structures are contiguous blocks containing a header describing the sizes of the array and its "strides" for efficient indexing, together with a variable-sized "value" block packed with floating point numbers. The most natural Rlisp representation is a vector of vector of floats; the floating point numbers, however, can end up scattered over the heap. In order to make the two representations more compatible, so that passing such a structure from PSL to C involved only a block copy from the PSL heap to the C malloc area, the floatArray package was developed.

FloatArrays are built on the lisp word vector construct. This discussion will not attempt to discuss the implementation details, but will try to cover the aspects of floatArray which are used in developing system code for objects based on floatArray. Note that the actual floatArray package provides a fairly low level of packed matrix access. Objects in Alpha_1 which are built on floatArrays always use a few conventions and common routines which are defined in "matrix.r". These provide some higher level functionality as a layer between routines which use those objects and the floatArray package itself. The "matrix.r" layer is described in the next section, but the conventions are described here.

FloatArrays basically implement a packed two-dimensional matrix structure. That is, each floatArray has some number of rows and columns. The base elements of the floatArray may be single floating point numbers (scalars), or any of the Alpha_1 point and vector types, including RnVecs. Thus, floatArray is quite flexible. One-dimensional arrays can be implemented by setting the number of rows to 1. A three-dimensional array of scalars can be implemented by setting the base type to be an rnVec of appropriate dimension. The Alpha_1 code uses floatArrays to implement knotVectors (one-dimensional scalar arrays), control polygons (one-dimensional arrays with any base type), control meshes (two-dimensional arrays with any base type), and others.

A floatArray is just a (very low-level) lisp word vector. So the functions which are defined on floatArray are what you will need to know about, rather than how the word vector is structured.

FloatArrays are created with

newFloatArray(*NumRows*, *NumCols*, *ElementObjType*)

Returns <floatArray> A new packed array of floating point values, initialized to zero.

NumRows <integer> The number of rows.

NumCols <integer> The number of columns.

ElementObjType
<keyword | integer> One of the point or vector object types: 'e2Pt, 'e3Pt, 'p2Pt, 'p3Pt, 'r2Vec, or 'r3Vec. It may also be **'SCALAR**, or a positive integer which is interpreted as the number of coordinates in an rnVec.

All of the elements of the floatArray are set to 0 and the header information completely initialized by **newFloatArray**. FloatArrays can be indexed using the standard Rlisp [] syntax. The routines which implement the indexing are described below.

The **floatArrayIndex1** extraction routine is not generally used (the conventions for handling rows will be described in the section on the matrix layer below). **FloatArrayIndex1** returns a floatArrayRow object which consists of the floatArray and an index field which has the specified index value. **FloatArrayIndex2** returns an object of the floatArray's base type and **floatArrayIndex3** returns a float.

floatArrayIndex1(*BlockPtr*, *Index*)

Returns <floatArrayRow> Extract a row of the floatArray.

BlockPtr <floatArray> The floatArray.
Index <integer> Which row you want.

floatArrayIndex2(*BlockPtr*, *Index1*, *Index2*)

Returns <element> Extract one of the elements of the floatArray (a point or vector, for example).
BlockPtr <floatArray> The floatArray to index.
Index1, *Index2*
<integer> Indices into the array.

floatArrayIndex3(*BlockPtr*, *Index1*, *Index2*, *Index3*)

Returns <float> A coordinate of the floatArray
BlockPtr <floatArray> The floatArray to index.
Index1, *Index2*, *Index3*
<integer > Indices into the array.

To assign values in the floatArray, a corresponding group of "set" functions are defined. To set a row of the array, use **setFloatArrayIndex1**. The others are **setFloatArrayIndex2** where ObjArg must match the base type of the floatArray, and **setFloatArrayIndex3** where *NumArg* is just a float.

setFloatArrayIndex1(*DestBlockPtr*, *Index*, *RowArg*)

Returns <floatArrayRow> The new values for the row.
DestBlockPtr
<floatArray> The float array.
Index <integer> Which row.
RowArg <floatArrayRow> The data to be put in the row. This may be a lisp list or lisp vector of objects of the correct base type. If the base type is scalar, the *RowArg* may be a point or a vector. The *RowArg* may also be any row object which follows the row object conventions described for the matrix layer at the end of this section (**ctlMeshRow** for example).

setFloatArrayIndex2(*DestBlockPtr*, *Index1*, *Index2*, *ObjArg*)

Returns <point | anyVector | number> The value which was assigned.
DestBlockPtr
<floatArray> The floatArray in which to put the value.
Index1, *Index2*
<integer> Index into the floatArray.
ObjArg <point | anyVector | number> The value to be assigned.

setFloatArrayIndex3(*DestBlockPtr*, *Index1*, *Index2*, *Index3*, *NumArg*)

Returns <float> The value which was assigned.
DestBlockPtr
<floatArray> The floatArray in which to put the value.
Index1, *Index2*, *Index3*
<integer> Index into the array.
NumArg <number> The value to be assigned.

Two routines are provided for extracting larger chunks of floatArrays, analogous to the lisp **subVector** routine. **SubFloatArray** extracts a consecutive group of rows from the floatArray, forming

them into another floatArray with the specified number of rows. **SubFloatArrayRow** returns a floatArray with just one row, and the specified number of columns:

subFloatArray(*FArray*, *BeginPos*, *Len*)

Returns	<floatArray> Extracts a group of rows from one floatArray, forming a smaller floatArray.
FArray	<floatArray> The floatArray.
BeginPos	<integer> Row index to begin extracting rows.
Len	<integer> How many rows are to be in the result.

subFloatArrayRow(*FArray*, *RowI*, *BeginPos*, *Len*)

Returns	<floatArray> A floatArray with just one row and the given number of columns.
FArray	<floatArray> The floatArray.
RowI	<integer> The index of the row from which the elements are to be extracted.
BeginPos	<integer> The column index to begin extracting elements.
Len	<integer> How many elements to extract; number of columns in the result.

A few other functions are useful for getting information about the floatArray other than its contents. To find out the base type which was specified when the floatArray was created with **newFloatArray**, use **floatArrayBaseObjType**. The sizes of the various floatArray dimensions are returned by three other functions.

floatArrayBaseObjType(*BlockPtr*)

Returns	<keyword | integer> Extract the base object type of the floatArray. It will be one of 'e2Pt, 'e3Pt, 'p2Pt, 'p3Pt, 'r2Vec, 'r3Vec, 'scalar, or a positive integer indiciating the dimension of an 'rnVec.
BlockPtr	<floatArray> The floatArray to be checked.

floatArrayNumberOfRows(*BlockPtr*)

Returns	<integer> The number of rows in the floatArray.
BlockPtr	<floatArray> The floatArray to be examined.

floatArrayNumberOfColumns(*BlockPtr*)

Returns	<integer> The number of columns in the floatArray.
BlockPtr	<floatArray> The floatArray to be examined.

floatArrayNumberOfCoordsPerElement(*BlockPtr*)

Returns	<integer> Number of coordinates in each base element of the floatArray.
BlockPtr	<floatArray> The floatArray to be examined.

Other functions in the floatArray package are for internal use only by the floatArray implementation files. The functions listed above are sufficient to implement objects based on floatArray using the matrix layer conventions described in "matrix.r".

Matrix Layer Conventions for Building Objects Based on FloatArrays

Each object based on floatArray in the Alpha_1 system should follow these conventions in order that standard procedures and accessing mechanisms can be used.

The first and most important convention is that the slot in the object which contains the floatArray is named "Block". Existing functionality could be duplicated without this restriction, but it would

drastically increase the number of methods which had to be defined for each new object. There are a few key functions which can be coded as generic procedures (not methods) if the mechanism for getting the float array is a simple slot accessor with a fixed name. These procedures can be applied to any object based on floatArray with this convention. **FloatArrayClassP** indicates whether the given object is based on floatArray.

floatArrayClassP(*ObjArg*)

Returns <boolean> If the object is based on floatArray objects, T is returned.
ObjArg <anything> The object to be checked.

baseType(*floatArrayObject*)

Returns <keyword | integer> Extract the base object type of the floatArray. It will be one of 'e2Pt, 'e3Pt, 'p2Pt, 'p3Pt, 'r2Vec, 'r3Vec, 'scalar, or a positive integer indiciating the dimension of an 'rnVec.
floatArrayObject
<floatArrayObj> The floatArray-based object to be examined.

numberOfRows(*floatArrayObject*)

Returns <integer> The number of rows in the floatArray object.
floatArrayObject
<floatArrayObj> The floatArray-based object to be examined.

numberOfColumns(*floatArrayObject*)

Returns <integer> The number of columns in the floatArray object.
floatArrayObject
<floatArrayObj> The floatArray-based object to be examined.

numberOfCoordsPerElement(*floatArrayObject*)

Returns <integer> The number of coordinates in the base element objects of the floatArray object.
floatArrayObject
<floatArrayObj> The floatArray-based object to be examined.

There are three standard methods which are provided for objects based on floatArray: **floatArrayClass**, **blockIndex**, and **objSize**. The floatArrayClass method just returns T so that the object can be identified as being based on floatArray: The **blockIndex** method always returns value 0. The **blockIndex** is used to implement row objects related to two-dimensional floatArray objects. These are described below as matrix row conventions. Until you are building a row object, don't worry about **blockIndex**; just define the method as shown. The **objSize** method returns the primary size of the object. If you are implementing one-dimensional arrays (where the number of rows is always 1), the method will probably just return the number of columns. For two-dimensional matrices, the method usually returns the number of rows. Again, this method has more significant implications when building row objects.

floatArrayClass(*Self*)

Returns <boolean> T if the object is in the floatArray class.
Self <anything> The object to be examined.

blockIndex(*Self*);

Returns <integer> Zero. (This is a dummy method for objects which don't need a

blockIndex field as described below).

Self <floatArrayObj> The object from which to retrieve the block index.

objSize(*Self*);

Returns **<integer> The size of the floatArray object in it's "major" dimension.**

Self **<floatArrayObj> The object to examine.**

It is usual to define two procedures for your object, which contain the name of the object or an abbreviation. One is a wrapper for the create function and **newFloatArray**, and the other is a wrapper for the **objSize** method. Examples for knotVector objects are **newKnotVector** and **kvSize**. The new function will call the creator for the object, providing the floatArray (Block) argument by calling **newFloatArray**. This gives you a chance to hide irrelevant arguments from the user level. The knotVector example is:

```
constructor procedure newKnotVector( Size );
     createKnotVector( newFloatArray( 1, Size, 'Scalar ) );
```

The size function will often use the **objSize** method:

```
extractor procedure kvSize( Kv );
     objSize Kv;
```

although for two-dimensional arrays, you may wish to copy the ctlMesh style:

```
extractor procedure cMeshSize( mesh, dir );
     matSize( mesh, dir );
```

The **matSize** procedure can be used on any floatArray object with a Block field. It returns the number of rows if *dir* is **ROW** and the number of columns otherwise.

You will want to define a number of **index** and **setIndex** methods which call appropriate **floatArrayIndex** and **setFloatArrayIndex** functions. This will allow Rlisp [] indexing to work on your new object. If you are defining an object in which the base element may be non-scalar (i.e., point or vector), you will also want to define one **indexCoord** method which allows "symbolic" access of the coordinates of the base elements (e.g., cPoly[0]->x, which is equivalent to cPoly[0][0]). The following table tells you which methods you will need and which **floatArrayIndex** functions they should call depending on the kind of array you are implementing. (The calling sequences for these are too tedious to list; check the source files for examples.)

If Your Object Is:	You Need Methods:	Which Calls floatArray:
one-dimensional, scalar elements only	index1	index2
one-dimensional, any element type	index1 index1Coord index2	index2 index3 index3
two-dimensional, scalar elements only	index1 index2	index1 * index2
two-dimensional, any element type	index1 index2 index2Coord	index1 * index2 index3

index3 index3

Since you want to be able to set values in the float array, as well as extract them, you will also need corresponding **setIndex** methods for each **index** method you define. The items marked with **asterisks indicate that you probably don't actually want to call floatArrayIndex1**, but rather should implement an additional row object as described in the section below on matrix row conventions.

The floatArray argument to the **floatArrayIndex** functions will always be "Block". The index methods for one-dimensional arrays will always fill in the first index in the **floatArrayIndex** call with a 0 (since there is only one row). The other parameters are just passed on, except for the **indexCoord** functions. These functions must translate the coordinate argument (the last index) into a numeric value appropriate for the base type of the floatArray using the function **arrayCoordIndex**. As an example, the **index1Coord** procedure for a ctlPoly is

```
method procedure CtlPoly index1Coord( Index1, Coord );
     floatArrayIndex3( Block, 0, Index1,
                       arrayCoordIndex( Block, Coord ) );
```

The same translation must occur in **setIndex1Coord**.

Finally, you may want to define a **subVector** method for your object to allow chunks to be extracted efficiently and conveniently. This method will call **subFloatArrayRow**, as shown in the ctlPoly example:

```
constructorMethod procedure CtlPoly subVector( BeginPos, Len );
   createCtlPoly( subFloatArrayRow( Block, 0, BeginPos, Len ) );
```

Matrix Row Conventions

If you are implementing a two-dimensional array type based on floatArray, you will probably want to define a floatArray row object that corresponds to the new object. The reason for this is to make the behavior of floatArray objects analogous to that of lisp "vector of vector" constructs. If you create a vector of vectors in lisp (called "mat" for example), then

```
row := mat[i];
```

will return a vector. In particular, "row" is a pointer into part of "mat". So changing an element of "row":

```
row[j] := NewValue;
```

is identical changing the element in "mat":

```
mat[i][j] := NewValue;
```

Conceptually then, a row object for a floatArray type (like ctlMeshRow for ctlMesh) is just a pointer into the block occupied by the matrix object. It is more complicated in implementation. The definition for a row object should be of the form:

```
defModelObject( ObjNameRow( !:creator ),
                Block( !:Type FloatArray ),
                RowNum( !:Type int ),
                BlockIndex( !:Type int ) );
```

The Block slot will hold a pointer to the original floatArray. RowNum will contain the index of the row of that floatArray that this object represents, and BlockIndex will contain a cached offset into the floatArray.

You will need to define the standard methods floatArrayClass and **objSize** as before. Note that **objSize** should return the number of columns in the original floatArray (Block). However, no

blockIndex method is needed since you have a slot by that name. Instead, provide a method called **numberOfRowsMethod** which always returns 1. A row object obviously has only one row, but since **floatArrayNumberOfRows** Block would return the number of rows in the floatArray which the row came from, we need another mechanism to override. The **numberOfRowsMethod** method is used in **setFloatArrayIndex1** to allow assignment of row objects into rows of floatArrays.

numberOfRowsMethod(*Self*)

Returns <integer> Number of rows in the floatArray object.
Self <floatArray> The floatArray-based object.

You should write a procedure for building a row object, given the array object and a row index. The Block field is just the Block field from the array object, and RowNum is just the index passed in. The BlockIndex is calculated from the row stride of the array, as shown for CtlMeshRow:

```
constructor procedure ctlMeshRow( CtlMeshArg, RowNumArg );
begin scalar BlockPtr, IndexTemp;

   BlockPtr := CtlMeshArg->Block;
   IndexTemp := RowNumArg * (floatArrayRowStride BlockPtr);
   return
      createCtlMeshRow( BlockPtr, RowNumArg, IndexTemp );
end;
```

You should also provide a procedure which invokes the **objSize** method to determine the size of the row object.

You will also need the **index** and **setIndex** methods. If the base type of the row elements is scalar, you need only **index1**. If it can be a point or vector type, you will need **index1**, **index1Coord**, and **index2**.

Finally, you must tie the new row object into the **index1** method for the original array object. If you have defined the row object as specified here, the details of the **setIndex1** method are taken care of by the floatArray package, so you just use:

```
method procedure ObjName setIndex1( Index, RowArg );
    setFloatArrayIndex1( Block, Index, RowArg );
```

where RowArg may be one of your row objects. The **index1** method should check the range and call the row object constructor, as shown for this ctlMesh example:

```
method procedure CtlMesh index1( Index );
   if Index < 0 or
         Index > ((floatArrayNumberOfRows Block) - 1) then
      rangeError( Self, Index, 'index1 )
   else
      ctlMeshRow( Self, Index );
```

8.7 Matrix & Mesh

The matrix and mesh packages are built on the **floatArray** package and provide a basic set of packed float array objects for use in **shape_edit**. They are good examples to follow ifyou need to build another **floatArray** class object.

The simplest vector and matrix objects are *mat1d* and *matrix*. Their creation and size functions

are **newVector**, **newMat**, **mat1dSize**, and **matSize**. (The name "vecSize" was already used for something else.)

newVector(*Size*)

Returns <mat1d> A new one-dimensional matrix.
Size <integer> Number of elements.

mat1dSize(*Vec*)

Returns <integer> Number of elements in the one-dimensional array.
Vec <mat1d> The object to be examined.

newMat(*RSize*, *CSize*)

Returns <matrix> A new matrix object.
RSize, CSize
<integer> Dimensions of the matrix.

matSize(*Mat*, *Dim*)

Returns <integer> Size of the matrix in the specified direction.
Mat <matrix> The object to be examined.
Dim <symbol> **ROW** or **COL** for which direction the size is desired for.

The indexing functions, **vecIndex** and **index**, are provided for compatibility with C and older code; you should just use Rlisp [] indexing. However, the **dirIndex** and **otherdir** functions are often useful for avoiding duplication of code in routines that have to do something in either the row or the column direction.

vecIndex(*Vec*, *Sub*)

Returns <float> The element from the one-dimensional matrix.
Vec <mat1d> The object from which the element is to be extracted.
Sub <integer> The index of the desired element.

index(*Mat*, *Rsub*, *Csub*)

Returns <float> The element from the matrix.
Mat <matrix> The object from which the element is to be extracted.
Rsub, Csub
<integer> Indices of the desired element.

otherdir(*D*)

Returns <integer> Index of the other direction (value of **ROW** or **COL**).
D <symbol | integer> **ROW** or **COL**, or 0 or 1. The other is returned.

dirIndex(*Mat*, *Dir*, *R*, *C*)

Returns <float> An element of the matrix.
Mat <matrix> The object from which the element is to be extracted.
Dir <symbol> **ROW** or **COL**, indicating which direction the indexing should be done in. The indexes will be swapped if *Dir* is **COL**.
R, C <integer> The indices of the desired element.

More direct access to **matSize** is available with **numRows** and **numCols**. Matrix multiplication (**matMult**) is provided for conformable matrices, and transposition is provided with **matTranspose**.

numRows(*Mat*)

Returns <integer> Number of rows in the matrix.
Mat <matrix> The object to be examined.

numCols(*Mat*)

Returns <integer> Number of columns in the matrix.
Mat <matrix> The object to be examined.

matMult(*Mat1*, *Mat2*)

Returns <matrix> Matrix multiplication of the two given matrices.
Mat1, Mat2
<matrix> The matrices to be multiplied.

matTranspose(*Mat*)

Returns <matrix> The transpose of the given matrix.
Mat <matrix> The matrix to be transposed.

The **knotVector**, **ctlPoly**, **ctlMesh**, and **ctlMeshRow** objects are all defined according to the floatArray conventions for matrices described in the previous section. They all define **floatArrayClass**, **blockIndex**, and **objSize** methods, together with appropriate indexing methods.

8.8 Basic Geometry

Almost all of the functions which implements basic geometry operations for **shape_edit** are intended to be user level functions, so they are described in the User's Manual. Many of the functions here are "helper" functions for the others, or are intended only to be used when implementing other system operations.

When **shape_edit** is built, a number of predefined geometric objects are initialized functions on the **Initializers!*** list. All of these functions take no arguments and are evaluated for side effect (i.e., they set global variables).

initLinesPckg
Sets **XAxis**, **YAxis**, **Zaxis**.

initPlanesPckg
Sets **XYPlane**, **XZPlane**, **YZPlane**.

initUnitCircle
Sets **UnitCircle**.

initLatLongSphere
Sets **LatLongSphere**.

initUnitSphere
Sets **UnitSphere**.

initCubeSphere
Sets **CubeSphere**.

initUnitCylinder
Sets **UnitCylinder.**

initUnitCubeWireFrame
Sets **UnitCubeWireFrame.**

initNominalVector
Sets **NominalVector!*.**

initLinDimDefaultAttrs
Sets **LinDimDefaultAttrs.**

initDiaDimDefaultAttrs
Sets **DiaDimDefaultAttrs.**

initAngDimDefaultAttrs
Sets **AngDimDefaultAttrs.**

initAnimationPckg
Sets **LinearTime**, **SlowIn**, **SlowOut**, **SlowInOut**, **BounceTime**, **SineTime**, **GlobalKineticCurve!***, and **GlobalKineticCurveName!*.**

initInteraction
Sets **PickTags!*.**

Functions that are most conveniently used with variable numbers of arguments are usually written as a macro wrapper with a procedure helper function. The procedure takes the arguments in a list. Some of the underlying procedures for these macro wrappers are listed below, together with the user level macro names.

ptL	**pt**
projPtL	**projPt**
vecL	**vec**
profileList	**profile**
crvConcatList	**crvConcat**
sameOrderList	**sameOrder**
srfFromCrvsList	**srfFromCrvs**
srfFromCrvsDirList	**srfFromCrvsDir**
objTransformL	**objTransform**
invObjTransformL	**invObjTransform**
transformL	**transform**
instanceL	**instance**
groupL	**group**
shell2	**shell**
crvTangLinesL	**crvTangLines**
crvTangLines2L	**crvTangLines2**

buildRestriction2
  buildRestriction
linDimAttrL
  linDimAttr
angDimAttrL
  angDimAttr
diaDimAttrL
  diaDimAttr

The **ptBlend** and **vecBlend** functions described in the User's Manual are actually part of a whole group of blending functions. Those two user level functions guarantee that the coefficients are all positive and sum to one. The **ptBlendLinear** and **vecBlendLinear** functions require the coefficients to sum to one, but they can be negative, and **ptBlendUnchecked** and **vecBlendUnchecked** don't check the coefficients at all. All of these functions eventually call **ptBlend2** and **vecBlend2** which actually to the blending. The **allPtBlend** group allows projective points to be blended (an operation which may be mathematically useful, but which doesn't make sense geometrically), and **scalarBlend** is for blending scalar values. The **checkCoefficients** routine is used throughout the blending functions wherever requirements are imposed on the coefficients.

ptBlendLinear(*PtList*, *CoeffList*)

ptBlendUnchecked(*PtList*, *CoeffList*)

ptBlend2(*PtList*, *CoeffList*, *CheckLevel*)

scalarBlend(*ScalarList*, *CoeffList*)

vecBlendLinear(*VecList*, *CoeffList*)

vecBlendUnchecked(*VecList*, *CoeffList*)

vecBlend2(*VecList*, *CoeffList*, *CheckLevel*)

allPtBlend(*PtList*, *CoeffList*)

allPtBlendLinear(*PtList*, *CoeffList*)

allPtBlendUnchecked(*PtList*, *CoeffList*)

allPtBlend2(*PtList*, *CoeffList*, *CheckLevel*)

checkCoefficients(*CoeffList*, *CheckFlag*)

Several "generic" operations are implemented as methods. The **bboxObj** operation invokes the bbox method, and **reverseObj** uses the **reverseOrient** method. The **reflect** operation uses the **reflectThru** method. For all the point types, this turns into a call on **reflectThruPt**, and for the two line types it turns into a call on **reflectThruLine**.

reflectThruLine(*Line*, *Obj*)

Returns	<object> Object reflected through the given line.
Line	<e2Line | e3Line> The line to reflect through.
Obj	<object> The object to reflect.

reflectThruPt(*Pt*, *Obj*)

Returns	<Object> Object reflected through the given point.
Pt	<point> The point to reflect through.
Obj	<object> The object to reflect.

The following functions are old and will eventually disappear, so don't use them: **bez2Crv**, **bez2KvP**, **bez2CrvP**, **checkBez2Crv**, **crvDivW**, and **srfDivW**.

8.9 Coercions

A package for coercion of objects containing points and vectors to other point and vector types is provided. It is not documented in the User's Manual because most routines coerce objects to the desired types as needed.

The basic routine in the package it **ptObjCoerce**, which makes all the points the specified type, following the same coercion rules described for points in the user's Manual. The **ptObjCommonType** function is used to find out the "minimal" point type that can be used for coercion of an object to a common point type without losing information. (E.g., if some of the points are E3 and some are E2, then E3 can be used. But if any are projective and any are 3D, then P3 is needed.) The **mkPtObjsCompatible** function combines the previous two functions into a single operation. The "ptVec" versions of the above allow vectors in the object to be coerced as well.

ptObjCoerce(*PtObj*, *Type*)

Returns	<object> Coerce all the embedded points in the object to the specified type.
PtObj	<object> The object containing the points to be coerced (a new copy will be returned).
Type	<keyword> The type of points desired: **'e2Pt**, **'e3Pt**, **'p2Pt**, or **'p3Pt**.

ptObjCommonType(*PtObj*)

Returns	<keyword> The type of the point which is the greatest common type for all the points in the object.
PtObj	The object containing the points.

mkPtObjsCompatible(*PtObj*)

Returns	<object> Coerce all the points in the object to the greatest common type.
PtObj	<object> The object containing the points to be coerced. (A copy of the object will be returned).

ptVecObjCoerce(*PtVecObj*, *Type*)

Returns	<object> Coerce all points and vectors in the object to the given type.
PtVecObj	<object> The object containing the points to be coerced.
Type	<keyword> The desired type for the points and vectors. It may be any point or vector type.

ptVecObjCommonType(*PtVecObj*)

Returns	<keyword> Determine the greatest common point type for all the points and vectors in the object.
PtVecObj	<object> The object containing the points to be examined.

mkPtVecObjsCompatible(*PtVecObj*, *VecCoerce*)

Returns	<object> Coerce all the points and vectors to the greatest common point type.
PtVecObj	<object> The object containing the points and vectors to be coerced.

VecCoerce <boolean> Relevant only if there are no points in the structure. If any points occur, all vectors and points are converted to the common point type. If the *VecCoerce* flag is Nil, vectors will be coerced to the highest dimension vector occuring in the structure. Otherwise, they will be coerced to points.

Methods which are used in the coercions functions are **ptCoerce**, **ptVecCoerce**, **commonType**, **vecToE2**, **vecToE3**, **vecToP2**, and **vecToP3**.

ptCoerce(*Self*, *Type*)

Returns <point> A point of the specified type.

Self <point> The point to coerce.

Type <keyword> The type to which the point should be coerced. The conversion between 2D and 3D is made by dropping the Z coordinate or adding a Z coordinate of zero. The conversion between projective and euclidean is made by dividing by the W coordinate or adding a W coordinate of 1.0.

ptVecCoerce(*Self*, *Type*)

Returns <point | anyVector> The point or vector coerced to the specified type.

Self <point | anyVector> The point or vector to be coerced.

Type <keyword> The desired type of the result. For vectors, the coercions between dimensions are made by dropping coordinates or adding coordinates of zero.

commonType(*Self*, *TypeFlags*, *Args*)

Returns <vector> The vector of *TypeFlags* passed in, modified if necessary.

Self <object> The object to be examined.

TypeFlags <vector> A vector of five values: point flag, projective flag, 3D flag, vector dimension, scalar flag. The point flag is true if a euclidean point has been found, the projective flag if a projective point has been found, and the 3D flag if any 3D points have been found. The vector dimension is the highest dimension of any vectors found so far, and the scalar flag is true if any curves or surfaces with scalar valued control meshes have been found.

Args <anything> Not used.

vecToE2(*Self*)

Returns <e2Pt> Construct an e2Pt from the given vector.

Self <anyVector> Vector to be converted.

vecToE3(*Self*)

Returns <e3Pt> Convert a vector into an e3Pt.

Self <anyVector> Vector to be converted.

vecToP2(*Self*)

Returns <p2Pt> Convert a vector to a p2Pt.

Self <anyVector> Vector to be converted.

vecToP3(*Self*)

Returns <p3Pt> Convert a vector to a p3Pt.

Self <anyVector> Vector to be converted.

Routines which help implement the package are described below.

typeCoercionFn(*Type*)

Returns <keyword> The name of a function for coercing points, vectors or scalars into the given type.

Type <keyword> The desired type.

toScalar(*Value*)

Returns <number> The same number (this is a dummy routine to avoid special cases).

Value <number> The value to be returned.

vecToPtCoercionFn(*Type*)

Returns <keyword> The name of a function for coercing vectors to a given point type.

Type <keyword> The desired point type.

coerceTo(*ptVecObj*, *ptVecType*)

Returns <point | anyVector> The coercion of a point or vector to the specified type.

ptVecObj <point | anyVector> The object to be coerced.

ptVecType <keyword> A point or vector type.

findCommonPtType(*PtVecObj*)

Returns <vector> The vector of flags filled in by the **commonType** methods. This is a helper function for **ptObjCommonType**, which initializes the flag vector.

PtVecObj <object> The object for which the common point type is to be found.

vecDimCoerce(*Vec*, *NewDim*)

Returns <anyVector> The coercion of the vector to the specified dimension.

Vec <anyVector> The vector to be coerced.

NewDim <keyword | integer> Desired dimension of the new vector (may be **'r2Vec** or **'r3Vec**).

8.10 Destructuring Operations

Two destructuring operations are provided which can help in writing other operations that need to deal with aggregates. *Aggregate* objects are those which are just collections of other objects. Examples include groups and shells. Aggregates have an **aggregateP** method which returns a true value. The **aggregate!-objs** functions finds out if an object is an aggregate.

aggregate!-obj(*Obj*);

Returns <boolean> Value is T if the object is an aggregate (group, instance, shell, combiner object).

Obj <anything> Object to be examined.

The two destructuring operations are **destructureObj** and **destructure2Obj**. The difference between them is in the way they handle their results. The **destructureObj** function accumulates a result in one of the arguments, while **destructure2Obj** forms a copy of the input objects structure, modified according to whatever actual operation is being applied. As examples, the **bboxObj** and **commonType** operations are based on **destructureObj**, while **raiseOrder**, **reverseObj**, and **pointApply** are based on **destructure2Obj**.

A set of input flags determine the actions that the destructuring operations take. The flags are stored in a 5-element list or lisp vector as follows:

Flag[0] Possible values are **'MapAndInvert**, **'Map**, and **'NoMap**. If the flag is not **'NoMap**, then objects which occur within instance objects are mapped through the appropriate transformations before the operations are applied. If **'MapAndInvert** is set, then the results are mapped back to their original space after the operations are applied. For **destructureObj**, the first two flags have the same effect (inversion of the transformation is not meaningful).

Flag[1] Possible values are **'ToPoints** and **'ToNonAggregates**. If **'ToPoints** is specified, objects containing points are destructured all the way down to the points. Otherwise, destructuring stops when non-aggregate objects are reached.

Flag[2] The value is the name of the method to apply when the bottom level of the destructuring is reached.

Flag[3] Possible values are **'ParamTypesHanded** and **'UseGeomField**. In the first case, parametric types are assumed to have a method to handle the operation. Otherwise, the geometry field of the parametric type will be destructured and the operations applied to the results.

Flag[4] Must be passed in as Nil. This is space reserved for the transformation stack for mapping and inverse mapping.

A set of wrapper functions make it easier for the destructuring routines to interpret the flag vector. Except for **methodToCall** and **destructureMats**, they all return a boolean value indicating the sense of the flags.

toPoints(*Flags*);

Returns <boolean> The value of *Flag*[1].

Flags <vector> The vector of destructuring flags.

destructureMap(*Flags*);

Returns <boolean> Whether the structure is to be transformed.

Flags <vector> The vector of destructuring flags. The function returns true unless the flag value is **'noMap**.

destructureInvMap(*Flags*);

Returns <boolean> Whether the structure is to be inverse transformed.

Flags <vector> The vector of destructuring flags. The function returns true if the value is **'MapAndInvert**.

useGeomField(*Flags*);

Returns	<boolean> The value of *Flags*[3].
Flags	<vector> The vector of destructuring flags.

destructureMats(*Flags*);

Returns	<transform | listOf matrixDescriptor> Objects representing the matrix stack accumulated so far in traversing the structure.
Flags	<vector> The vector of destructuring flags.

methodToCall(*Flags*);

Returns	<keyword> The name of the method to call on bottom-level objects.
Flags	<vector> The vector of destructuring flags.

The destructuring operations themselves use methods **destructureM** and **destructure2M** to do their work.

destructureObj(*Obj*, *Flags*, *RetVal*, *Args*);

Returns	<anything> The value accumulated in *RetVal* by the method calls.
Obj	<anything> The object to destructure and apply the given method to.
Flags	<vector> The flag vector, as described above.
RetVal	<anything> The initial value for the return value.
Args	<list> Any arguments which are to be passed on to the methods.

destructureM(*Self*, *Flags*, *RetVal*, *Args*);

Returns	<anything> The value of *RetVal* as returned by applying the method to the objects within the structure.
Self	<object> The aggregate object which is to be destructured.
Flags	<vector> The flag vector.
RetVal	<anything> The accumulated result value.
Arg	<list> Any arguments which are to be passed on to the methods.

destructure2Obj(*Obj*, *Flags*, *Args*);

Returns	<object> A copy of the object which has had the specified method applied to its elements.
Obj	<object> The object to destructure and apply to the method to.
Flags	<vector> The flag vector, as described above.
Args	<list> Any arguments to be passed on to the methods.

destructure2M(*Self*, *Flags*, *Args*);

Returns	<object> A new copy of the object, the elements of which have had the method applied.
Self	<object> The object to be destructured.
Flags	<vector> The flag vector, as described above.
Args	<list> Any extra arguments to be passed on to the methods.

The **pointApplyObj** routine is a call to **destructure2Obj** with flag values that are common for shape operators:

```
'[ MapAndInvert, ToPoints, pointApply, UseGeomField, nil ]
```

The **pointApply** method is provided for all the point types. It applies a specified procedure to the points.

pointApplyObj(*ObjStruct*, *Fn*, *Args*);

Returns	<object> A copy of the object, with the specified method applied.
ObjStruct	<object> The object to be destructured and on which the operation will be performed.
Fn	<keyword> The name of the function to call on the elements.
Args	<list> Any extra arguments to be passed along to the elements.

pointApply(*Self*, *Args*);

Returns	<object> The result of applying the first element of *Args* (a function name), to the object.
Self	<object> The object on which to operate.
Args	<list> A list of the function to apply, together with any arguments needed.

8.11 Spline Geometry

A large set of functions for dealing with splines are available below the user level in Alpha_1. These functions manipulate knot vectors and perform a veriety of spline evaluation, refinement, and degree-raising operations.

8.11.1 Knot Vector Generation

Given a set of control points, there are a number of alternative methods which may be used to generate an appropriate knot vector. From the user level, the choice can be made by indicating a keyword such as **KV_UNIFORM** or **KV_CHORD** for the knot vectory type. The routines **uniformKnotVec**, **bezierKnotVec**, **chordKnotVec**, and **angularKnotVec** generate knot vectors for **KV_UNIFORM**, **KV_BEZIER**, **KV_CHORD**, and **KV_ANGULAR** respectively.

uniformKnotVec(*CrvSrfFit*, *Dir*, *IgnoreArgs*)

Returns	<knotVector> A uniform knot vector appropriate for the given data.
CrvSrfFit	<curve | surface | fitSpec> The data for which the knot vector is to be generated.
Dir	<symbol> **ROW** or **COL** for surfaces, irrelevant for others.
IgnoreArgs	<Nil> Not used.

bezierKnotVec(*CrvSrfFit*, *Dir*, *IgnoreArgs*)

Returns	<knotVector> A Bezier knot vector appropriate for the given data. The Bezier knot vector has interior knots of multiplicity (order-1).
CrvSrfFit	<curve | surface | fitSpec> The data for which the knot vector is to be generated.
Dir	<symbol> **ROW** or **COL** for surfaces, irrelevant for others.
IgnoreArgs	<Nil> Not used.

chordKnotVec(*CrvSrf*, *Dir*, *IgnoreArgs*)

Returns	<knotVector> A chord length knot vector appropriate for the given data. This style of knot vector spaces knots proportionately to control point spacing.
CrvSrfFit	<curve | surface | fitSpec> The data for which the knot vector is to be generated.
Dir	<symbol> **ROW** or **COL** for surfaces, irrelevant for others.
IgnoreArgs	<Nil> Not used.

angularKnotVec(*CrvSrf*, *IgnoreDir*, *CtrList*)

Returns	<knotVector> An angular knot vector appropriate for the given data. This style of knot vector spaces knots proportionately to angular control point spacing about a center point.
CrvSrfFit	<curve | fitSpec> The data for which the knot vector is to be generated.
Dir	<Nil> Not used.
CtrList	<listof Point> The single argument is the center point (in a list). If Nil, the origin is used as the center.

The **chordKnotVec** routine separates its work into a surface case and a curve case. For surfaces, it uses the average chord length along successive rows or columns of the control mesh. The angular knot vector type uses **makeAngleKv**.

makeCrvChordKv(*Curve*)

Returns	<knotVector> A chord length knot vector for a curve.
Curve	<curve> The curve for which the knot vector is to be generated.

makeSrfChordKv(*Srf*, *Dir*)

Returns	<knotVector> A chord length knot vector for one direction of a surface.
Srf	<surface> The surface for which the knot vector is to be generated.
Dir	<symbol> The direction for the knot vector.

makeChordPars(*PtVector*, *EcFlag*)

Returns	<knotVector> A knot vector structure containing the normalized parameter values at which the control points occur.
PtVector	<ctlPoly> The control polygon.
EcFlag	<symbol> End condition flag (need to wrap around if it is **EC_PERIODIC**).

formKvFromPars(*ParametricValues*, *EcFlag*

Returns	<knotVector> The knot vector which is constructed from a list of parameter values computed in **makeChordPars**.
ParametricValues	<knotVector> The list of parameter values indicating desired spacing of the knots (usually computed from the control point spacing).
EcFlag	<symbol> End condition flag of the curve for which the knot vector is being generated.

makeAngleKv(*Curve*, *Center*)

Returns	<knotVector> An angularly spaced knot vector for the curve.

Curve <curve> The curve for which the knot vector is to be computed.
Center <point> The center point, from which angles will be measured.

The **getKValue** function accesses the knot vector slot of a curve or surface parminfo and checks to see if a knot vector has been generated yet. If not, an appropriate knot vector is generated using one of the generators described above.

getKvValue(*Parms, Dir, CrvSrfFit*)

Returns <knotVector> The value of the knot vector for the given curve, surface, or fitting specification.
Parms <parmInfo | listof parmInfo> The parametric information for the object.
Dir <symbol> **ROW** or **COL** for surfaces, irrelevant otherwise.
CrvSrfFit <curve | surface | fitSpec> The object for which the knot vector is being accessed or generated.

8.11.2 Knot Vector Utilities

The routines described in this package are often useful in building higher-level operators which have to deal with the parametric (knot vector) aspect of spline curves and surfaces.

The **knotIndex** routine locates the index of a particular knot in the knot vector and returns this integer. If the knot does not occur (and it must occur exactly because floating point equality is used), then Nil is returned. The **knotIndex** routine has one very important application. In general, adding knots of multiplicity order-1 produces evaluation points at the knot values. For spline curves and surfaces with **OPEN** end conditions, the index of the evaluation point in the refined control polygon or mesh is exactly the index of the occurrence of the knot in the refined knot vector. So, given a list of parametric values at which evaluation points are desired, use the **multKnots** procedure to produce a refinement knot vector, refine the curve or surface, and use the **knotIndex** procedure to determine indices for extracting the evaluation point from the surface.

knotIndex(*KnotVal, KnotVect*)

Returns <integer | Nil> Index of the given knot in the knot vector.
KnotVal <number> The value to be checked for.
KnotVect <knotVector> The knot vector to search in.

The **Multknots** routine constructs a refinement knot vector for evaluating a curve or surface at a set of parametric values. These values are assumed to be in knot-vector order (i.e., the sequence is non-decreasing), and should not have any repetitions. (This actually makes the previous requirement strictly increasing). The knot vector returned contains knots to be merged in which cause a multiple knot to exist at each parametric value after refinement. The routine insures that the knots will not exceed the maximum allowed order. **Warning**: There are possible side effects. Numerical instabilities will occur if one of the requested values is **very** close, but not exactly equal to an existing knot. In this case, the requested value vector is **modified** by this routine so that the requested value is exactly on the existing knot. This happens only when the values are quite close, and so it should not affect the accuracy of the result. But it may cause problems in application code if there are other (not updated) copies of the knot values around.

multKnots(*KnotVals, KnotVect, Order*)

Returns <knotVector> A refinement knot vector which will cause the resulting curve or surface to have an evaluation point at each requested value.
KnotVals <listOf number> The values to be checked for.

KnotVect <knotVector> The knot vector being searched.
Order <integer> The order associated with the knot vector.

The **addMult** function appends a knot of the specified value and multiplicity to the end of the given knot vector. The **addMultInVect** routine adds a knot of the specified multiplicity to the knot vector passed in, returning a larger vector. As opposed to **addMult**, this routine allows insertion of the knot inside the vector (rather than only allowing appending the value on the end).

addMult(*Val*, *Mult*, *Kv*)

Returns <listOf number> A knot of the specified value and multiplicity is added to the knot list.
Val <number> Knot value to be added.
Mult <integer> Multiplicity of the knot to be added.
Kv <listOf number> Knot list to which the multiple knot is to be appended.

addMultInVect(*Val*, *Mult*, *Kv*)

Returns <knotVector> A new knot vector with a knot of the given value and multiplicity inserted.
Val <number> The value to be added.
Mult <integer> Multiplicity of the knot to be added.
Kv <knotVector> Knot vector to which the value is to be added.

The **checkKnot** function examines a knot vector to determine whether a specific knot value occurs, and if so how many times it occurs (the multiplicity of the knot).

checkKnot(*KnotValue*, *KnotVect*)

Returns <Nil | dottedPairOf number> The multiplicity of the occurence of the value in the knot vector, as described below.
KnotValue <number> The value to be checked for.
KnotVect <knotVector> The knot vector looking in.

There are three reasonable return values:

Nil The value is not present.
(Nil . ***m*** **)** The value is present with multiplicity **m**.
(***n*** **.** ***m*** **)** The value is not present, but one which is very close occurs with multiplicity **m**. The value of the nearly equal knot is **n**.

This routine is the one which is used by the **multKnots** routine (above), so if the noted side effect of that routine causes problems, this routine will probably be helpful in constructing code to circumvent them.

To normalize a knot vector, use **kvNormalize**. A normalized knot vector is one which has 0 as its first value and 1 as its final value. Any knot vector can be normalized (via translation and scaling) without changing the curve which it partially defines. This is often useful as a pre-process when stringing together a set of independently defined curves. The routine **kvNormalizeInPlace** normalizes the knot vector without making a copy.

kvNormalize(*Kv*)

Returns <knotVector> Normalized so that first knot is 0 and last knot is 1.

Kv	<knotVector> The knot vector to be normalized.

kvNormalizeInPlace(*Kv*)

Returns	<knotVector> The normalized knot vector. The knot vector is knot copied first.
Kv	<knotVector> The knot vector to normalize.

The nodes of a curve can be computed using **computeNodes**. A vector of the nodes of the B-spline curve (which actually depend only on the knot vector and the order) is returned. The number of nodes (for open end conditions, which is probably the only case for which this works) is exactly equal to the number of control points in the curve.

computeNodes(*Parm*)

Returns	<vectorOf number> A vector of the nodes for the given parametric information.
Parm	<curve | parmInfo> Object for which to compute the nodes.

Merging two knot vectors is accomplished with **joinKvs**. Both knot vectors are assumed to have open end conditions. The resulting knot vector has a single knot of multiplicity order-1 in place of the two order-fold knots which were in the original knot vectors. The second knot vector is shifted appropriately.

joinKvs(*Order*, *Kv1*, *Kv2*)

Returns	<knotVector> Concatenated version of two knot vectors, with a knot of multiplicity order-1 at the join.
Order	<integer> Order of the two knot vectors.
Kv1, Kv2	<knotVector> The knot vectors to be concatenated.

This is an important step in merging two curves. If the endpoints coincide, then the curve control polygons may simply be concatenated together, with the duplicate endpoint dropped.

For a list of the distinct knots in the knot vector of a curve, **distinctKnots** returns a **list** of the knot vector, except for the multiple values at the beginning and end (e.g., if cKv Crv is [0 0 0 1 2 2 3 3 3], then **distinctKnots** will return (0 1 2 2 3)). Open end conditions are assumed.

distinctKnots(*Crv*, **DistinctFlag**)

Returns	<listOf number> A list of the distinct knots in a knot vector.
Crv	<curve> The curve for which distinct knots are being extracted.
DistinctFlag	<boolean> If T, each multiple knot in the original is listed once in the result. If Nil, only the multiple knots at the beginning and end of the knot vector are removed.

Two methods are important for being able to access the various parminfo fields without having to know what kind of object the parminfo is contained in. These are **parmSelect** and **sizeOfSplineMeshInDir**.

ParmSelect(*Self*, *Dir*, *Fn*)

Returns	<integer | symbol | knotVector> The requested parmInfo field.
Self	<object> The object from which the parmInfo field is to be extracted.
Dir	<integer | symbol> Depends on the object, but some way to specify from which parmInfo the field is to be extracted.

Fn <keyword> Which field is wanted.

sizeOfSplineMeshInDir(*Self*, *Dir*)

Returns	<integer> Size of the mesh in the given direction.
Self	<object> The object for which the size is to be computed.
Dir	<integer | symbol> Depends on the object, but some way to specify for which direction the field is to be extracted.

8.11.3 End Conditions

Three kinds of spline end conditions are supported in Alpha_1: *open*, *floating*, and *periodic*. By far the most frequently used is the open end condition spline, which touches the first and last control points and is tangent to the control polygons at those points (like a Bezier curve). There are conversion functions for just about any combination you would want. The higher level functions are described in the User's Manual, so theones here are low level helper functions.

surfaceOpenInDir(*Surface*, *Dir*)

surfFloatingFromPeriodic(*Surface*)

floatFromPerInDir(*Surface*, *Dir*)

perFromFloatInDir(*FSrf*, *Dir*, *PerKv*)

surfOpenfromFloating(*Surface*)

openFromFloatInDir(*Surface*, *Dir*)

curveFloatingfromPeriodic(*Curve*)

curveOpenfromFloating(*Curve*)

curvePeriodicFromFloating(*Fcrv*, *PerKv*)

Several knot vector utilities are used for these end condition conversions.

unrollPerKv(*PeriodicKv*, *Order*)

Returns	<knotVector> An equivalent floating knot vector for the periodic one.
PeriodicKv	<knotVector> The knot vector to be converted.
Order	<integer> Associated polynomial order.

unrollKv(*KnotVector*, *Left*, *Right*)

Returns	<knotVector> An equivalent floating point knot vector for the periodic one.
KnotVector	<knotVector> The knot vector to be converted.
Left, Right	<integer> The number of knots on either end by which to unwrap the knot vector.

checkFloatKv(*OldKv*, *NewKv*, *Order*)

Returns	<knotVector> Remove any knots that define extraneous basis functions.
OldKv	<knotVector> Original knot vector.
NewKv	<knotVector> New refinement knot vector.
Order	<integer> Polynomial order associated with the knot vectors.

unrollPoly(*CtlPolygon*, *Index*, *NewPoints*)

Returns <ctlPolygon> Extended polygon for treatment as equivalent floating curve.
CtlPolygon
<ctlPolygon> The original (periodic) control polygon.
Index <integer> Index where the wrapping around begins.
NewPoints
<integer> Number of new points to add.

8.11.4 Control Meshes

The basic surface, curve, and knot vector constructors attempt to figure out almost any reasonable specification of a control mesh. The **vectorFromBunch** and **listFromBunch** functions form lisp vectors and lists from "bunches" of point values.

vectorFromBunch(*Bunch*)

Returns <vector> A lisp vector datastructure.
Bunch <Nil | vector | list | pair> The structure to be converted.

listFromBunch(*Bunch*)

Returns <list> A lisp list datastructure.
Bunch <Nil | vector | list | pair > The structure to be converted.

A basic check at construction time with **checkParmMeshCompat** insures that surfaces and curves have the correct relationships between order, number of knots, and number of control points. Control points are coerced to the "smallest" possible point type determined by **crvSrfCommonType**.

checkParmMeshCompat(*ParmInfo*, *MeshSize*, *FnName*)

Returns <Nil> If any incompatibilities between the control mesh and the parametric information are found, an error is signaled.
ParmInfo <parmInfo> The parametric information.
MeshSize <integer> Size of the mesh (or control polygon) in the associated direction.
FnName <keyword> Name of the function from which the checking routine was called.

crvSrfCommonType(*CrvList*)

Returns <keyword> Common point or vector type for a list of surfaces and curves.
CrvList <listOf surface | curve> The list of objects to be examined.

Occasionally, it is useful to be able to insert entire rows or columns of control points into a surface mesh using **crvToSrf**. The **grabCrvPts** function gets a subset of the control points in a curve as a list.

grabCrvPts(*PtList*, *StartIndx*, *EndIndx*)

Returns <vector> A subset of the points.
PtList <vector | list | ctlPolygon> The structure from which to extract the points.
StartIndx, *EndIndx*
<integer> Indices of where to start and stop.

crvToSrf(*Crv*, *Srf*, *Dir*, *Index*)

Returns	<surface> Control points from the curve are copied into the specified row or column of the surface.
Crv	<curve> The curve to be inserted.
Srf	<surface> The surface into which the points are to be copied.
Dir	<symbol> **ROW** or **COL** to specify which direction to copy.
Index	<integer> Index of the row or column into which to copy.

8.11.5 Refinement

Refinement in Alpha_1 uses the Oslo algorithm, which computes an "alpha" matrix describing the blending of old (original) control points to form a new (refined) set of control points for the curve. The **newAlpha** reoutine sets up a new alpha matrix to use in refinement. The **calcAlpha** function fills in the alpha matrix, and the **crvAlphaMul** and **srfAlphaMul** functions actually use the alpha matrix to compute the new set of curve and surface control points. The higher level functions, **lispCRefine** and **lispSRefine**, perform refinement of curves and surfaces respectively, using **calcAlpha** and either **crvAlphaMul** or **srfAlphaMul**. The **cRefine** and **sRefine** routines described in the User's Manual actually call the C implementation of the refinement algorithm, and are preferred. Finally, **cRefine** and **sRefine** normally check the input data to make sure it has "reasonable" numbers of knots and control points. Occasionally, if you really know what you are doing and are familiar with the details of the Oslo algorithm, you may want to override this check for special kinds of refinement using **cRefineNoCheck** or **sRefineNoCheck**.

newAlpha(*order*, *nrows*, *ncols*, *ColAlloc*)

Returns	<alphaMat> A new alpha matrix.
order	<integer> Order of the knot vector for which it will be used.
nrows, ncols	<integer> Size of the matrix.
ColAlloc	<boolean> Whether to allocate the columns now, or defer it until later.

calcAlpha(*EC_Flag*, *Order*, *KvTau*, *KvT*)

Returns	<alphaMat> The alpha matrix for refinement.
EC_Flag	<symbol> End condition of the knot vector.
Order	<integer> The order of the knot vector.
KvTau	<knotVector> The old knot vector.
KvT	<knotVector> The new knot vector.

CrvAlphaMul(*EC_Flag*, *OldPoly*, *alpha*, *BegPos*, *len*)

Returns	<ctlPolygon> A control polygon refined according to the alpha matrix given.
EC_Flag	<symbol> End condition type of the curve.
OldPoly	<ctlPolygon> Original control polygon.
alpha	<alphaMat> The alpha matrix to use for refinement.
BegPos	<integer> Index in the control polygon of where to start computing.
len	<integer> How many new points to calculate.

srfAlphaMul(*EcFlag*, *OldMesh*, *dir*, *alpha*, *BegPos*, *len*)

Returns	<ctlMesh> The control mesh refined according to the alpha matrix given.
EcFlag	<symbol> End condition type of the surface in the specified direction.

OldMesh <ctlMesh> Original control mesh.
dir <symbol> **ROW** or **COL**, indicates which direction to refine in.
alpha <alphaMat> The alpha matrix to use for refinement.
BegPos <integer> Index in the control mesh of where to start computing.
len <integer> How many new points to calculate.

lispCRefine(*OldCrv*, *NewKnots*, *DoMerge*)

Returns <curve> Refined curve, computed in Lisp code rather than passing to the C routines.
OldCrv <curve> Original curve.
NewKnots <listOf number | knotVector> The refined knot set.
DoMerge <boolean> Whether *newKnots* needs to be merged into the original knot vector (not necessary if the new knot vector already contains all the original knots).

lispSRefine(*OldSrf*, *Dir*, *NwKnots*, *DoMerge*)

Returns <surface> Refined surface, computed in Lisp code rather than passing to the C routines.
OldSrf <surface> Original surface.
Dir <symbol> **ROW** or **COL**, indicating the direction in which the refinement should be done.
NwKnots <listOf number | knotVector> The refined knot set.
DoMerge <boolean> Whether *newKnots* needs to be merged into the original knot vector (not necessary if the new knot vector already contains all the original knots).

sRefineNoCheck(*OldSrf*, *Dir*, *NwKnots*, *DoMerge*)

Returns <surface> Refined surface, without compatibility checking. (This can be reasonable, if you really know what you are doing.)
OldSrf <surface> Original surface.
Dir <symbol> **ROW** or **COL**, indicating the direction in which the refinement should be done.
NwKnots <listOf number | knotVector> The refined knot set.
DoMerge <boolean> Whether *newKnots* needs to be merged into the original knot vector (not necessary if the new knot vector already contains all the original knots).

cRefineNoCheck(*OldCrv*, *NwKnots*, *DoMerge*)

<curve> Refined curve, without compatibility checking. (This can be reasonable, if you really know what you are doing.)
OldCrv <curve> Original curve.
NwKnots <listOf number | knotVector> The refined knot set.
DoMerge <boolean> Whether *newKnots* needs to be merged into the original knot vector (not necessary if the new knot vector already contains all the original knots).

8.11.6 Degree Raising

The degree raising routines described in the User's Manual use **raiseOrderObj** to destructure aggregate objects until curves and surfaces are reached. The **raiseOrderM** method for curves and surfaces actually does the degree raising computations. For surfaces, the curve routine **incrementOrderCrv** is just applied once for each row or column (not very efficiently at this point). The **findRValue** routine is a helper for **incrementOrderCrv**.

raiseOrderObj(*Obj*, *CrvSrfFlag*, *DirFlag*, *Order*)

Returns <object | listOf object | vectorOf object> Perform degree-raising on elements in the structure.
Obj <object | listOf object | vectorOf object> The structure on which to perform degree-raising.
CrvSrfFlag
<keyword> If **'Srf**, only surfaces are degree-raised. If **'Crv**, only curves are degree-raised. Otherwise, both are done.
DirFlag <symbol> Direction to perform degree-raising for surfaces (Nil implies both directions to be done.)
Order <number> The desired order.

raiseOrderM(*Self*, *ArgList*)

Returns <object> A new copy of the object, possibly degree-raised.
Self <object> The object to be degree-raised.
ArgList <list> List of the remaining flags as described for **raiseOrderObj** above.

IncrementOrderCrv(*CurveIn*)

Returns <curve> A new curve of one degree higher than the original.
CurveIn <curve> The curve on which to operate.

findRvalue(*NewKnots*, *OldKnots*, *J*)

Returns <integer> An index into the old knot vector, for curve degree-raising.
NewKnots, *OldKnots*
<knotVector> The new knot vector and old knot vector.
J <integer> Index into the new knot vector.

8.11.7 Spline Evaluation

The derivative calculation in **diffCrv** uses **diffCrvOnce** and **crvNthDerivative** to calculate derivative splines. Two additional evaluation functions, **leftHandCrvEval** and **crvNthDerivLeftEval** are provided for evaluating curves from left hand limites rather than the default right had limits. These evaluation functions actually use an implementation of the Cox-DeBoor algorithm to do the evaluation. The **coxDeBoor** function has two helper functions: **computeKnotIndex** for finding the index at which to base the evaluation, and **coxDeBoorInternalFn**.

diffCrvOnce(*Crv*)

Returns <curve> The curve representing the first derivative.
Crv <curve> The curve to be differentiated.

crvNthDerivative(*Crv*, *N*)

Returns <curve> A curve representing the Nth derivative.
Crv <curve> The curve to be differentiated.
N <integer> The number of derivatives to take.

leftHandCrvEval(*Crv*, *Tee*)

Returns <point | anyVector | scalar > Evaluation of the curve (the left hand limit if there are internal discontinuities).
Crv <curve> The curve to be evaluated.
Tee <number> The parametric value at which to evaluate the curve.

crvNthDerivLeftEval(*Crv*, *N*, *Tee*)

Returns <point | anyVector | scalar > Evaluation of the Nth derivative of the curve (the left hand limit if there are internal discontinuities in the Nth derivative.)
Crv <curve> The curve to be evaluated.
N <integer> The number of derivatives to take.
Tee <number> The parametric value at which to evaluate the curve.

coxDeBoor(*Crv*, *Tee*, *LimitFlag*)

Returns <point | anyVector | scalar> Evaluate the curve from the left or right limit for the given parametric value.
Crv <curve> The curve to be evaluated.
Tee <number> The parametric value at which to evaluate the curve.
LimitFlag <keyword> For left hand limits, should be **'Left**, otherwise use **'Right**.

computeKnotIndex(*Order*, *Knots*, *Tee*,*LimitFlag*)

Returns <integer> Index of the knot on which to base a Cox-deBoor evaluation.
Order <integer> Order of the curve.
Knots <knotVector> The knot vector of the curve.
Tee <number> Value at which the evaluation is being performed.
LimitFlag <symbol> **'Left** or **'Right**, as in **coxDeBoor**.

coxDeBoorInternalFn(*Order*, *Points*, *Knots*, *Tee*, **KnotIndex**)

Returns <point | anyVector | scalar> Evaluation point of a curve.
Order <integer> Order of the curve.
Points <ctlPolygon> Relevant points from the curve (copied, this operation is destructive).
Knots <knotVector> The knot vector for the curve.
Tee <number> The parametric value at which to evaluate the curve.
KnotIndex
<integer> Index in the knot vector at which to base the calculation.

8.11.8 Miscellaneous Geometry

A group of functions which compute the differential geometry information for a *Frenet frame* is used in several applications, most notably N/C toolpath computations.

The **stepFrenet** routine computes a series of Frenet coordinate systems at specified parameter values on the curve and returns these in a list. The curve may be of arbitrary order but must

use open end conditions. The tangent, normal, and binormal are computed by **frenetCoords** from the first three control polygon points for the curve evaluated at the point of interest. It returns a coordinate system which represents the Frenet frame. The **tangNormBinorm** routine returns a list containing the tangent, normal, and binormal vectors of the curve defined by interpreting *PtList* as (at least) the first three points of a Bezier curve. The returned values are **not** normalized, since the magnitudes may be important for some operations.

stepFrenet(*Crv*, *ParamList*, *moveToOrigin*)

Returns <listOf list> Compute Frenet frames for a curve at specified points. The list of Frenet frames (as lists of three vectors) and the list of evaluation points are returned.
Crv <curve> Spline curve.
ParamList <listOf number> List of parameter values at which Frenet frame is to be calculated.
moveToOrigin
<boolean> If true indicates that the Frenet frame coordinate systems are to be based at the evaluated curve point, rather than the origin.

frenetCoords(*P0*, *P1*, *P2*, *InvertFlag*, *MoveToOrigin*)

Returns <matDescr> A matrix descriptor for the Frenet frame.
P0, P1, P2
<point> First three Bezier control points at curve evaluation point.
InvertFlag <boolean> Should be true if the Bezier control points come from the **end** of the curve instead of the **beginning**. The flag causes the direction of the tangent vector to be inverted.
MoveToOrigin
<boolean> If true Frenet frame origin is based at *P0* (the evaluation point), rather than the origin.

tangNormBiNorm(*PtList*)

Returns <listOf geomVector> The tangent, normal, and binormal computed from the three given points.
PtList <listOf point> The three points at beginning of a Bezier curve segment.

FrenetFrame(*Pt1*, *Pt2*, *Pt3*, *TangFlip*)

Returns <listOf geomVector> Normalizes the result of **tangNormBiNorm** (which it calls).
Pt1, Pt2, Pt3
<point> The three points at the beginning of a Bezier curve segment.
TangFlip <boolean> Whether or not to flip the tangent and binormal vectors.

computeFrenetFrame(*Crv*, *CrvDerivs*, *Param*)

Returns <frenetFrameObj> Compute the Frenet frame for a point on a curve.
Crv <curve> The curve to be evaluated.
CrvDerivs <listOf curve> The list of curves returned by **crvDerivSet**.
Param <number> Parameter value at which to calculate the Frenet frame.

8.12 Primitives

Most of the primitives used in the primitives package are quite simple to create as spline surfaces, and so the **makeGeom** methods for each primitive do all the work. One curve type which is not predefined or available as a primitive is the ellipse. These are constructed with **createEllipse** and are used by the ellipsoid primitive. createEllipse

createEllipse(*Maj, Min*)

Returns	<curve> A curve representing an ellipse.
Maj, Min	<geomVector> Vectors describing the major and minor axes of the ellipse.

Some of the other primitives use **cylTopologyAdj** and **boxTopologyAdj** to create the adjacency declarations for the surfaces that represent the primitives.

cylTopologyAdj(*SurfList*)

Returns	<Nil> Generate adjacencies for objects with the same topology as a cylinder.
SurfList	<listOf surface> The list of surfaces which make up the object.

boxTopologyAdj(*SurfList*)

Returns	<Nil> Generate adjacencies for objects with the same topology as a box.
SurfList	<listOf surface> The list of surfaces which make up the object.

The rounded box primitives are a much more complex entity and there are a number of routines used to create the spline surface representation.

IOU - Rounded Box Primitives (esc)

8.13 Groups & Instances & Transforms

The **flattenTree** function is used by the shell constructor to insure that the geometry of a shell is a pure list of surfaces, not containing any group or instance objects. Using a helper function, **flattenTree2**, it applies whatever transformations are required to make a single level group object representing the same geometry.

flattenTree2(*ObjStruct, Mat, FlatList*)

Returns	<listOf object> Flattens the structure, returning a list of objects. None of the remaining items are lists, vectors, groups, or instances.
ObjStruct	<object | listOf object | vectorOf object> The structure to be flattened.
Mat	<matrix4x4> The current matrix accumulated from instances in the structure. Initialize to Nil the first time.
FlatList	<listOf object> The list of objects being accumulated for the result. Initialize to (Nil . Nil).

Similarly, the constructor for groups insures that no embedded lists or lisp vectors are in the input arguments using **destructure**. (Other groups and instance objects may be embedded in a group). Note that this function is not related to **destructureObj**.

destructure(*Struct*)

Returns	<listOf object> A list of the objects with no embedded lists or vectors.
Struct	<object | listOf object | vectorOf object> The structure to collect objects from.

A number of methods are defined on transforms and the individual matrix descriptor objects to implement the various matrix descriptor list editing functions.

matDescrPMethod(*Self*)

Returns	<boolean> True value is always returned if this method is defined.
Self	<matDescr> The object. The method is only defined for matrix descriptor objects.

extractMatrix(*Self*)

Returns	<mat4x4> The 4x4 matrix that results from concatenating all the matrix descriptors in the object.
Self	<object> The object for which the matrix is to be computed.

appendDescriptor(*Self*, *DescrArg*)

Returns	<Nil> Appends a new matrix descriptor to the list of matrix descriptors in an instance or transform.
Self	<object> The object to which the new descriptor is to be added.
DescrArg	<matDescr> The new matrix descriptor.

insertDescriptor(*Self*, *N*, *DescrArg*)

Returns	<Nil> Inserts a new matrix descriptor in the list of matrix descriptors in an instance or transform.
Self	<object> The object to which the new descriptor is to be added.
N	<integer> Index of the item after which the new one will be put.
DescrArg	<matDescr> The new matrix descriptor.

deleteDescriptor(*Self*, *N*)

Returns	<Nil> Deletes a matrix descriptor in the list of matrix descriptors in an instance or transform.
Self	<object> The object from which the descriptor is to be deleted.
N	<integer> Index of the item after which one will be deleted.

replaceDescriptor(*Self*, *N*, *DescrArg*)

Returns	<Nil> Replaces a matrix descriptor in the list of matrix descriptors in an instance or transform.
Self	<object> The object in which the descriptor is to be replaced.
N	<integer> Index of the item to be replaced.
DescrArg	<matDescr> The new matrix descriptor.

indexDescriptor(*Self*, *N*)

Returns	<matDescr> The Nth matrix descriptor in the list of matrix descriptors in an instance or transform.
Self	<object> The object from which the descriptor is to be retrieved.
N	<integer> Index of the desired descriptor.

The **matDescrP** procedure uses the **matDescrPMethod** functions. In order to compute an explicity 4x4 matrix representing the matrix descriptor list, **extractMatrix** uses **concatOnRight**, **concatDescrL**, **matDescrFromPt**, and **matrix4x4P**.

matDescrP(*Obj*)

Returns <boolean> True if the object is a matrix descriptor.
Obj <object> The object to be tested.

concatOnRight(*DescrArg*, *Mat*)

Returns <mat4x4> Concatenate the matrix representing a matrix descriptor onto an existing matrix.
DescrArg <matDescr> The matrix descriptor.
Mat <mat4x4> The matrix.

concatDescrL(*DescrList*)

Returns <mat4x4> The matrix specified by the sequence of matrix descriptors.
DescrList <listOf matDescr | point | mat4x4 > The list of items to concatenate into a single matrix.

matDescrFromPt(*PtArg*)

Returns <matDescr> A matrix descirptor for the translation specified by a point.
PtArg <point> The point.

matrix4x4P(*Arg*)

Returns <boolean> True if the object is a 4x4 matrix.
Arg <mat4x4> The matrix to be tested.

The **objTransform** and **invObjTransform** routines use their own destructuring operation rather than **destructureObj**. The **mapObjStruct** routine breaks up lists and lisp vectors, and it calls **mapObject** to handle objects, each of which should have provided a **mapObj** method. Objects which don't provide **mapObj** methods are simply copied by **objTransform**. This is nice in that things don't blow up when new objects are added, and dangerous because you don't necessarily know that what you want is not happining. The **checkMatrix** and **checkDiagonalMatrix** functions are utilities which help determine the type of points which may result from the mapping (and hence what dimension resulting packed arrays will be).

mapObjStruct(*PtStruct*, *Mat*)

Returns <listOf object | vectorOf object | object> Transform the input structure.
PtStruct <listOf object | vectorOf object | object> The structure to be transformed.
Mat <mat4x4> The matrix to be used for the transformation.

mapObject(*ObjArg*, *Mat*)

Returns <object> Push an object through the transformation.
ObjArg <object>The object.
Mat <mat4x4> The matrix.

mapObj(*Self*, *Mat*)

Returns <object> The transformed object.
Self <object> The object to be transformed.
Mat <mat4x4> The matrix.

checkMatrix(*Mat*,*RotAxisRestr*,*TransAxisRestr*)

Returns <number | Nil> Indicates whether the matrix meets certain restrictions. If so, it returns the uniform scale value. Otherwise it returns Nil. The

matrix may not contain perspective transformations, skew transformations, or differential scales.

Mat <mat4x4> The matrix.

RotAxisRestr
<symbol> 'X, 'Y, or 'Z (or Nil) to indicate which rotation axis restrictions must be maintained. The specified axis must not move or change sign.

TransAxisRestr
<symbol> 'X, 'Y, or 'Z (or Nil) to indicate which translation axis restrictions must be maintained. No translations may occur along the specified axis.

checkDiagonalMatrix(*Mat*)

Returns <number | Nil> Check whether the upper 3x3 of the matrix is a scalar times the identity matrix. If it is, return the scale value, otherwise return Nil.

Mat <mat4x4> The matrix.

The **invObjTransform** function actually uses the same code as **objTransform**; it just creates the inverse matrix descriptor list on the way in using the **createInverseDescr** methods provided by the matrix descriptor objects.

createInverseDescr(*Self*)

Returns <matDescr> A matrix descriptor which represents the inverse of the one given.

Self <matDescr> The descriptor to invert.

8.14 Combiner Objects

IOU - Combiner Objects (esc)

8.15 Attribute Package

The user level **getAttr**, **setAttr**, and **remAttr** use a number of low level routines to implement the operations. The **attrP** function can be used to determine whether a given attribute exists for an object. The **addAttr** function is used by **setAttr**, which checks that the new attribute is not duplicated before calling **addAttr**.

setAttr2(*AttrList*, *AttrName*, *Value*)

Returns <aList> Replace an entry in an attribute list, possibly adding new sublists.
AttrList <aList> The current attribute list.
AttrName <keyword> The name of the attribute.
Value <anything> The value associated with this attribute.

remAttr2(*AttrList*, *AttrName*)

Returns <aList> Remove an entry in an attribute list or sublist.
AttrList <aList> The current attribute list.
AttrName <keyword> The name of the attribute to remove.

getAttr2(*AttrList*, *AttrName*)

Returns <anything> Get an entry from an attribute list or sublist, returning Nil if the attribute is not present.
AttrList <aList> The attribute list.
AttrName <keyword> The name of the attribute.

addAttr(*Obj*, *AttrName*, *Value*)

Returns <aList> Add an attribute without deleting an existing attribute with the same name.
Obj <object> The object to which the attribute is to be added.
AttrName <keyword> The name of the attribute.
Value <anything> The value of the attribute.

addAttr2(*AttrList*, *AttrName*, *Value*)

Returns <aList> Add an attribute to the aList, without deleting existing attributes with the same name.
AttrList <aList> The attribute list.
AttrName <keyword> The name of the attribute.
Value <anything> The value of the attribute.

getBoundAttr(*Obj*, *AttrName*)

Returns <any | '!*Unbound!*> Get the requested attribute of the object, or return **'!*Unbound!*** if the attribute is not present.
Obj <object> The object from which the attribute is to be retrieved.
AttrName <keyword> The name of the attribute.

getBoundAttr2(*AttrList*, *AttrName*)

Returns <any | '!*Unbound!*> Get the requested attribute from the attribute list, or return **'!*Unbound!*** if the attribute is not present.
AttrList <aList> The attribute list.
AttrName <keyword> The name of the attribute.

AttrP(*Obj*, *AttrName*)

Returns <boolean> Returns true if the object has the specified attribute.
Obj <object> The object.
AttrName <keyword> The name of the attribute to check for.

AttrP2(*AttrList*, *AttrName*)

Returns <boolean> Returns true if the attribute list contains the specified attribute.
AttrList <aList> The attribute list to check.
AttrName <keyword> The attribute name to check for.

AddAttrsL(*Obj*, *OptionsPile*)

Returns <object> The object to which the attributes were added.
Obj <object> The object to which the attributes are to be added.
OptionsPile
<list> A list of alternating names and values which will all be added to the attribute list of the object as name-value pairs.

8.16 Adjacency Package

IOU - Adjacency Package (esc)

8.17 Shape Operators

A number of the surface shaping operations are based on the **pointApply** operations described earlier (see section 8.10 [Destructuring Operations], page 144). All of them provide a procedure to be applied to points in the input data structure. See the User's Manual for descriptions of the arguments, as they are passed straight through from the top level.

bend2(*CtlPt*, *Axis*, *OtherAxis*, *BendCenter*, *MinRange*, *MaxRange*, *BendRate*)

flattenSrf2(*CtlPt*, *P,Dir,Dot,SFlg,RestrictL*)

warp2(*CtlPt*, *DirectionVector*, *CenterPt*

regionWarp2(*CtlPt,DirectionVector*, *WarpFactor*

skelWarp2(*CtlPt*, *DirectionVector*, *WarpFactor*

stretch2(*CtlPt*, *XScale*, *YScale*, *ZScale*)

taper2(*CtlPt*, *TaperFn*, *Ax*, *OtherAx,ExtraArgs*)

twist2(*CtlPt*, *Ax*, *Ax1*, *Ax2*, *TwistFn*, *Args*)

The warping operations use some extra functions for determining what points to move and how far to move them. The **vecMeasure** function measures distance in a specified norm (L2 for Euclidean, other are possible). That distance is used in the **warpWeight** function to decide how far to move a particular control point. The weighting functions use **floatExp** to do floating exponent calculations.

vecMeasure(*P1*, *P2*, *Norm*)

Returns	<number> The distance between to points using the specified L(n) norm.
P1, *P2*	<point> The points.
Norm	<integer> The exponent for the measure. Value 2 gives the usual euclidean norm.

warpWeight(*Distance*, *WarpFactor*, *Cutoff*)

Returns	<number> A number between 0 and 1 indicating the weighting for moving a point in the warp operator as a function of its distance from the base point.
Distance	<number> Distance from the warp base point.
WarpFactor	<number> An exponent (roughly) for the weighting function.
Cutoff	<number> The distance at which the weight must fall to zero.

floatexp(*x*, *y*)

Returns	<number> Computes *x* to the *y* power. (Lisp **exp** function only allows *y* to be an integer.)
x, *y*	<number> Base and exponent, respectively.

The skeletal warp uses **skelMeasure** to measure diatance from a point to a polyline. The **skelPreprocess** function is used by both **skelWarp** and **regionWarp** to precompute information about the

skeleton. (For **regionWarp** it is called from **regionPreprocess**.) The **regionMeasure** routine measures distance from a point to the region boundary for **regionWarp**, if the point was determined to lie outside the region in **ptInsideRegion**.

skelMeasure(*CtlPt*, *Skel*, *N*)

Returns	<number> Compute the distance from a point to a polyline.
CtlPt	<point> The point to measure distance to the skeleton.
Skel	<list> A special list of values as computed by **skelPreprocess** for a polyline.
N	<integer> Measuring norm to use, where 2 is the euclidean (L2) norm.

skelPreprocess(*Skel*, *CloseFlg*)

Returns	<list> A special form optimized for measurements to be done in **skelWarp** and **regionWarp**.
Skel	<polyline | polygon | curve | listOf point | vectorOf point> Something that can be interpreted as a polyline.
CloseFlg	<boolean> Whether the region needs to be closed.

regionPreprocess(*Region*)

Returns	<list> A special form optimized for measurements to be done in <polyline | polygon | curve | listOf point | vectorOf point> Something that can be interpreted as a polyline.

regionMeasure(*CtlPt*, *Region*, *N*)

Returns	<number> Compute the distance from a point to the region.
CtlPt	<point> The point to compute distance for.
Region	<list> The region, in the special form computed by **regionPreprocess**.
N	<integer> The measuring norm to use.

ptInsideRegion(*CtlPt*, *Region*)

Returns	<boolean> True if the point is inside the region.
CtlPt	<point> The point to check.
Region	<list> The region, in the special form computed by **regionPreprocess**. Although not enforced anywhere in the code, the region must lie in the XY plane for this to work.

The edge blending operations **singleSrfEdge** and **srfEdge** each decompose to a separate function for each possible order of the cross section.

singleLinearSrfEdge(*Srf1*, *Edge1*, *Srf2*, *Edge2*)

Returns	<surface> Construct a linear surface between the edges of two other surfaces.
Srf1, Srf2	<surface> The two surfaces.
Edge1, Edge2	<keyword> The corresponding edge specifications (e.g., 'LEFT).

singleQuadraticSrfEdge(*Srf1*, *Edge1*, *Srf2*, *Edge2*, *Tang*)

Returns	<surface> Construct a quadratic surface between the edges of two other surfaces. The result will be tangent continuous with the first surface, and position continous with the second one.

Srf1, Srf2 <surface> The two surfaces.
Edge1, Edge2
<keyword> The corresponding edge specifications (e.g., '**LEFT**).
Tang <number> A scaling factor for the cross-boundary tangent vectors.

singleCubicSrfEdge(*Srf1, Edge1, Srf2, Edge2, Tang*)

Returns <surface> Construct a cubic surface between the edges of two other surfaces, tangent continuous to both of them.
Srf1, Srf2 <surface> The two surfaces.
Edge1, Edge2
<keyword> The corresponding edge specifications (e.g., '**LEFT**).
Tang <number | listOf number> Scaling factors for the cross-boundary tangent vectors on both surfaces. If a single number is given, it is used for both of the surfaces.

linearSrfEdge(*TopSrf, BotSrf*)

Returns <surface> A linear surface joining all four boundaries of two surfaces.
TopSrf, BotSrf
<surface> The two surfaces to be joined.

cubicSrfEdge(*TopSrf, BotSrf, TangDistInfo*)

Returns <surface> A cubic surface joining all four boundaries of two surfaces.
TopSrf, BotSrf
<surface> The two surfaces to be joined.
TangDistInfo
<number | listOf number> Scaling factors for the cross-boundary tangents of each of the surfaces. If a single number is given, it is used for both surfaces.

In addition, the **srfEdge** version (which goes around the corners of the surface) uses **checkSrfCorners** to locate any degenerate corners where the U and V tangents lie on the same line so that the corner blend generation code will be skipped at those places. The **getBoundaryTang** function computes a cross boundary tangent vector at each control mesh point on the edge. These can be used in **hermiteSrfFromCrvs** to make the blend surface. The surfaces for each boundary and edge are merged with **joinSrfListRow**. The **srfNoNulls** routine is just a little function that removes null elements from the list before the merging.

checkSrfCorners(*Srf*)

Returns <Nil> Determine whether any of the corners of a surface are degenerate by having three co-linear points. Set an attribute of the surface with the results.
Srf <surface> The surface to be checked.

getBoundaryTang(*Srf, Edge, CrnrCheck*)

Returns <curve> A curve representing the cross-boundary tangents for a given boundary of the surface.
Srf <surface> The surface to compute tangents for.
Edge <keyword> Which boundary to compute, e.g., '**TOP**.
CrnrCheck
<boolean> Whether to examine the attribute of the surface which describes

the result of checking for degenerate corners.

hermiteSrfFromCrvs(*Bnd1*, *Tang1*, *Tang2*, *TangScale*)

Returns <surface> Construct a surface from Hermite curve information: two position curves and two tangent curves.

Bnd1, Bnd2
<curve> The boundary curves with positional data.

Tang1, Tang2
<curve> The tangent curves with vector data.

TangScale <number> Scaling factor for the tangents.

joinSrfListRow(*SrfList*)

Returns <surface> Join a list of surfaces in the row direction. No checks are done for compatability.

SrfList <listOf surface> The surfaces to be joined.

srfNoNulls(*SrfList*)

Returns <listOf surface> A list of surfaces with no embedded Nil values.

SrfList <listOf surface> The list of surfaces.

The **lift** and **offset** operations are aided by low level operations **crvLift**, **srfLift**, **offset2**, **cOffset2**, **vOffset2**, and **cvOffset2**.

crvLift(*C*, *DirectionInfo*, *DistanceInfo*)

Returns <curve> Perform a lifting of a curve (does the work for **cLift**).

C <curve> The curve to modify.

DirectionInfo
<point> Direction of the lifting. If Nil, normals from the curve are used.

DistanceInfo
<number | vectorOf number | keyword> Description of the distance to move a point. May be a constant, a table lookup, or a function call (where the name of the function is given).

srfLift(*S*, *DirectionInfo*, *DistanceInfo*)

Returns <surface> Perform a lifting of a surface (does the work for **lift**).

S <surface> The surface to modify.

DirectionInfo
<point> Direction of the lifting. If Nil, normals from the surface are used.

DistanceInfo
<number | table | keyword> Description of the distance to move a point. May be a constant, a table lookup, or a function call (where the name of the function is given).

offset2(*S*, *Dist*)

Returns <surface> Compute approximation of offset to a surface.

S <surface> The surface to offset.

Dist <number> The offset distance.

cOffset2(*C*, *Dist*)

Returns <curve> Compute appoximation of offset to a curve.
C <curve> The curve.
Dist <number> Offset distance.

vOffset2(*S*, *DistInfo*)

Returns <surface> Compute a variable offset of a surface.
S <surface> The surface to offset.
DistInfo <number | table | keyword> Same interpretation as for **srfLift** above.

cvOffset2(*C*, *DistInfo*)

Returns <curve> The variable offset of a curve.
C <curve> The curve to offset.
DistInfo <number | vectorOf number | keyword> Same interpretation as for **crvLift** above.

Several of the shape operators that do computations relative to a coordinate axis use **nextAxis** and **prevAxis** to decide which coordinates to measure with or modify, and **axisCoord** to turn symbolic coordinate references into index values.

nextAxis(*Axis*)

Returns <keyword> The next coordinate axis (in the order **'X**, **'Y**, **'Z**).
Axis <keyword> The current axis.

prevAxis(*Axis*)

Returns <keyword> The previous coordinate axis (in the order **'X**, **'Y**, **'Z**).
Axis <keyword> The current axis.

axisCoord(*Axis*)

Returns <integer> The coordinate index of the specified axis.
Axis <keyword> The axis to convert.

The restriction regions used by the **flatten** and **warp** operations are actually implemented with calls to **checkPtRegion**, **checkPtPlane**, and **checkPtCoord**. The **regionProject** function is used to project a polyline onto one of the coordinate planes.

checkPtRegion(*Pt*, *Region*, *Proj*, *InOutFlag*)

Returns <boolean> Whether or not the point is inside (or outside) the region.
Pt <point> The point to test.
Region <list> The region to test the point against.
Proj <keyword> Projection in which to do the measuring (the test only works in 2D). Should be one of **'XY**, **'YZ**, or **'XZ**.
InOutFlag <boolean> If true, test for the point being outside rather than inside the region.

checkPtPlane(*Pt*, *Plane*, *PositiveNegativeFlag*)

Returns <boolean> Decide whether a point is on the positive or negative side of a plane.
Pt <point> The point to test.
Plane <plane> The plane to compare with.

PositiveNegativeFlag
<boolean> If true, test for the point being on the negative side of the plane.

checkPtCoord(*Pt*, *WhichCoord*, *CoordVal*, *LessGreaterFlag*)

Returns <boolean> Decide whether a particular coordinate of a point is less (or greater) than a given coordinate value.
Pt <point> The point to test.
WhichCoord
<integer> Which coordinate to test against.
CoordVal <number> The coordinate value to test with.
LessGreaterFlag
<boolean> If true, test whether the coordinate is is greater than the given value rather than less.

regionProject(*Region*, *Proj*)

Returns <list> A special form of the region which makes measurements simpler later. The region is projected onto the specified coordinate plane.
Region <curve | polyline | polygon | listOf point | vectorOf point> The region to process.
Proj <keyword> Coordinate plane on which to project the region (one of **'XY**, **'YZ**, or **'XZ**).

The refinement operations **addFlex**, **featureLine**, and **isolateRegion**, all try to make a correspondence between a coordinate axis and one of the parametric surface directions. They use **axisCorrespondence** and **coordCorrespondence** for this purpose. If the correspondence can be made, the choice of the region to refine is made by **findParametricRegion** or **findCrvParametricRegion**.

findParametricRegion(*S*, *Axis*, *MinVal*, *MaxVal*, *RefCrv*)

Returns <listOf number> The two parameter values on the curve which best fit the coordinate range given.
S <surface> The surface being measured.
Axis <keyword> Which axis to measure along (e.g., **'X**).
MinVal, MaxVal
<number> The desired range measured along the axis.
RefCrv <boolean> If true, measure against the curve on the opposite boundary from what the default would have chosen.

axisCorrespondence(*S*, *Axis*)

Returns <symbol | Nil> Decides whether the **ROW** or **COL** direction of the surface best corresponds to the given coordinate axis. If a decision is not clear, Nil is returned.
S <surface> The surface.
Axis <keyword> The axis of interest (e.g., **'X**).

coordCorrespondence(*C*, *Axis*, *Value*, *Epsilon*)

Returns <number> The parametric value on the curve which corresponds to a particular coordinate along the specified axis.
C <curve> The curve.
Axis <keyword> The axis to measure along.

Value <number> The coordinate value for which a corresponding parametric value is to be found.

Epsilon <number> How close the result must be.

findCrvParametricRegion(*C*, *Axis*, *MinVal*, *MaxVal*)

Returns <listOf number> The two parameter values on the curve which best fit the coordinate range given.

C <curve> The curve being measured.

Axis <keyword> Which axis to measure along (e.g., **'X**).

MinVal, *MaxVal*
<number> The desired range measured along the axis.

8.18 Interpolation Package

The User's Manual describes several curve interpolation routines which are built on the much more general curve and surface interpolation package described here.

8.18.1 General Curve Interpolation

Not all uses of curve interpolation can be made into packaged routines like those described in the User's Manual. To meet more general needs, there is a way to specify and solve more general interpolation problems. This general form is considerably more complex than the specialized routines. Therefore, it is preferable to use the specialized routines whenever possible. All the specialized routines above are implemented using the general mechanism. The general mechanisms described in the remainder of this chapter are fairly cumbersome, and a casual or beginning user is advised to avoid them if possible.

The first step is to create a fitting specification or *FitSpec*. Once a FitSpec has been created, a call to **crvInterp** will produce the curve (if it exists):

crvInterp(*FitSpec*)

Returns <curve> The curve which interpolates the given data.

FitSpec <fitSpec> Data for curve interpolation. (Described below.)

CrvInterp returns the unique curve (if it exists) that satisfies the fitting specification.

The remaining topics in this section describe elements of a fitting specification.

Fitting Specifications

A FitSpec contains complete information specifying a curve or surface. There are two parts to a FitSpec: the *parmInfo* and a number of *dataItems*:

fitSpec(*ParmInfo*, *DataSpec1*, ...)

Returns <fitSpec> Creates a fitting specification object for interpolation data.

ParmInfo <parminfo> A spline parmInfo for fitting (described below).

DataSpec1, ...
<dataItem> Calls to **dataItem**.

The **degreesOfFreedom** function returns an integer value that is the number of degrees of freedom in the linear system corresponding to its argument, and **numberOfConstraints** returns an integer value that is the number of constraints in the linear system corresponding to its argument.

degreesOfFreedom(*FitSpec*)

Returns <integer> The number of degrees of freedom in the interpolation data.
FitSpec <fitSpec> The interpolation data.

numberOfConstraints(*FitSpec*)

Returns <integer> The number of constraints in the interpolation data.
FitSpec <fitSpec> The interpolation data.

A necessary (but not sufficient) condition for the interpolation to succeed is that the degrees of freedom equal the number of constraints.

Fitting Parminfo

The first part of a FitSpec is a parmInfo (an object containing parametric information about a spline). For curve interpolation the fitting parmInfo specifies the **parmInfo** of the resulting curve. For surface interpolation it specifies the parmInfo in the U direction for the resulting surface.

This parmInfo may be any of the usual parmInfos for splines. However, there is an additional knot vector generator (**KV_COMPLETE**) that may be used for FitSpecs, but not for curves or surfaces. The **KV_COMPLETE** keyword only works in conjunction with an open end condition (**EC_OPEN**). The presence of this end condition is assumed (and not checked).

Without modification, **KV_COMPLETE** causes a knot to be placed at each parametric value corresponding to a position (as indicated by *BasisFnId* of position in a dataItem). This knot vector is so named because it is suitable for complete cubic curve interpolation. In other interpolation problems it might not provide the correct number of degrees of freedom. The degrees of freedom and the number of constraints of a FitSpec must be equal for interpolation to be possible. Use the functions **degreesOfFreedom** and **numberOfConstraints** to check.

For many interpolation problems, it is desirable to use a modification of a complete knot vector. For instance, it may be useful to be able to remove knots to lower the degrees of freedom, or to insert multiple knots where discontinuities are desired. For this reason two knot vector modifiers are implemented, **KV_Mod_NotAKnot** and **KV_Mod_AddAKnot**. They only work for **KV_COMPLETE**. For knot vector modification, the ID **KnotLocs** is bound to a vector containing all the parameters corresponding to position interpolation. The modifiers accept RLisp expressions involving **KnotLocs**. This is analogous to the use of the ID **ParamLocs** in the creation of dataItems.

The following example provides a parmInfo for fitting a cubic curve through positions without specifying end tangents. This is accomplished by specifying that the two parameters next to the ends are not knots.

```
P := parmInfo( CUBIC, EC_OPEN,
               KV_COMPLETE( KV_Mod_NotAKnot(
                                  KnotLocs[1],
                                  KnotLocs[size(KnotLocs)-1])));
```

In the following example, `KnotLocs[3]` is made a double knot, and `KnotLocs[1]` is not a knot.

```
P := parmInfo( CUBIC, EC_OPEN,
               KV_COMPLETE( KV_Mod_NotAKnot( KnotLocs[1] ),
                            KV_Mod_AddAKnot( KnotLocs[3] )));
```

DataItam

A dataItem specifies information to be interpolated. It contains an indication of what information is being interpolated, a parameter location where that interpolation is taking place, the data to be interpolated, and keywords to specify how the data is to be interpreted.

The constructor for the dataItem type is the function **dataItem**. It returns a list of dataItems, one for each parameter location.

dataItem(*BasisFnId*, *DataFnId*, *ParamLocExpr*, *DataBlock*)

Returns <dataItem> An object representing one piece of data for interpolation.
BasisFnId <symbol> Specifies what kind of information to be interpolated. (Described below.)
DataFnId <symbol> Specifies how the data is to be interpreted. (Described below.)
ParamLocExpr
<number | expr> Specifies at what parameter value the interpolation is to be performed. It is a number, or an expression involving the magic variable **ParamLocs**.
DataBlock <crvData | srfData> An instance of the crvData (or srfData) data type.

The *BasisFnId* may be one of the following:

Position
Tangent
LeftTangent
Derivative(N)
LeftDerivative(N)

The *BasisFnId* specifies what linear property is being interpolated. For example, if **ParamLoc** equaled 1.2, having a *BasisFnId* of Derivative(3) would put a constraint on the value of the third derivative evaluated at 1.2. The **LeftTangent** and **LeftDerivative** refer to computing a limit from below at discontinuities, rather than the default, which is to compute the limit from above.

The *DataFnId* may be one of the following:

ParamData
EstTangent
EstTangentUsingSecants

The *DataFnId* specifies how the data contained in the crvData (or srfData) is to be interpreted. Note that for surface interpolation, only **ParamData** is implemented.

The **ParamData** keyword matches the value in the *ParamLoc* slot with a parameter in the param slot of the dataBlock. It indicates that the corresponding datum in the Value slot of the dataBlock is the data value to be interpolated. This is the most commonly used *DataFnId*.

The **EstTangent** and **EstTangentUsingSecants** keywords indicate that tangent estimators are to be applied to the dataBlock to derive a value to be interpolated. These are available for curve interpolation only. **EstTangent** fits a polynomial curve through the data, and differentiates it at the place specified by the **ParamLoc**. **EstTangentUsingSecants** computes secants between the data points, fits a polynomial through these secants, and extrapolates that to the **ParamLoc** to generate a tangent estimate. In both cases the degree of the interpolatory polynomial is governed by the number of parameters in the param slot of the dataBlock (which is the same as the number of data values in the Values slot).

It would be possible to create dataItems one at a time. But because that can be very tedious (since so many dataItems will be needed in the typical interpolation problem), a mechanism is available to create several dataItems at once by referring to the information contained in the dataBlock. This is accomplished by binding the "magic" variable **ParamLocs** to a vector containing the parameters in the dataBlock. So the third argument to dataItem may be an expression involving the variable **ParamLocs**. Any functions that can be used to access lisp vectors can be used on **ParamLocs**.

Particularly useful are size and vector indexing.

The *ParamLocExpr* evaluates to a number, or a list or vector of numbers. The numbers specify at what parametric location the interpolation is to take place. A list of dataItems will be returned by dataItem, one for each of the numbers produced by the *ParamLocExpr* expression.

For surface interpolation, **ParamLoc** specifies a U value (where a pair of parameters (U, V) specifies a point on the surface).

The following statements construct crvDatas for the extensive examples given at the end of this section.

```
PositionCrvData := crvData( list( 0, 1, 2, 3, 4, 5, 6, 7 ), PosData );

TanData := vector( vec( .4, -.85 ), vec( .2, -.35 ));
TangentCrvData := crvData( list( 0, 7 ), TanData );
```

In the example below, DataItem1 will be a list containing a single dataItem, using only the first parameter, while DataItem2 uses only the last parameter. A list containing several dataItems, one for each parameter is constructed for DataItem3 and DataItem4.

```
DataItem1 := dataItem( Position, ParamData, ParamLocs[0],
                       PositionCrvData );

DataItem2 := dataItem( Position, ParamData, ParamLocs[size ParamLocs],
                       PositionCrvData );

DataItem3 := dataItem( Position, ParamData, ParamLocs,
                       PositionCrvData );
DataItem4 := dataItem( Tangent, ParamData, ParamLocs, TangentCrvData );
```

It would be possible to insert a parameter directly without using the variable **ParamLocs**. However, it is best to localize the specific information in the creation of the crvData (or srfData). Furthermore, using the **ParamLocs** mechanism guarantees that the **ParamLoc** slot value in the resulting dataItem is a parameter contained in the DataBlock. Should the **ParamLoc** slot value fail to be in the resulting DataItem, the interpolation will not always succeed.

The function **getDataValue** takes a dataItem as argument and returns the data value it specifies.

getDataValue(Data)

Returns	<number | point | geomVector> Value of the dataItem.
Data	<dataItem> The data to get the value of.

How would we create a dataItem that would use position information to estimate a tangent? We need to keep in mind that we probably want to use some subset of the information that is contained in the DataBlock, (since use of high degree polynomial interpolation to estimate the tangent is in general bad). The solution involves using the **subCrvData** (or **tailCrvData**) functions described in the crvData section below, as well as an appropriate **ParamLocs** expression.

For this example let's use the **EstTangent** tangent estimator keyword. To estimate an initial tangent using 3 positions, we could say:

```
DataItem5 := dataItem( Tangent, EstTangent, ParamLocs[0],
                       subCrvData( PositionCrvData, 0, 3 ));
getDataValue( first( DataItem5 ));
```

To estimate a final tangent using 4 positions, we could say:

```
DataItem6 := dataItem( Tangent, EstTangent,
                       ParamLocs[size ParamLocs],
                       tailCrvData( PositionCrvData, 4 ));
getDataValue( first( DataItem6 ));
```

CrvData

A CrvData contains fitting data for a curve. Its creator is **crvData**.

crvData(*Param*, *Values*)

Returns	<crvData> An object representing the interpolation data for a curve.
Param	<listOf number | vectorOf number> An ordered list or vector of numbers.
Values	<listOf point | listOf geomVector | vectorOf point | vectorOf geomVector > List or vector of points or vecs, one for each of the *Param* numbers. Each entry in *Values* should be of the same type.

For example, we could use positional data or tangent data:

```
PositionCrvData :=
    crvData( list( 0, 1, 2, 3, 4, 5, 6, 7 ), PosData );

TanData := vector( vec( .4, -.85 ), vec( .2, -.35 ));
TangentCrvData := crvData( list( 0, 7 ), TanData );
```

The function **subCrvData** returns a CrvData that is a subset of the first argument. This is useful for providing a subset of positional information to a tangent estimator. A similar routine, **tailCrvData**, returns a CrvData containing the last several parameters and values in its first argument. This is easier to use than **subCrvData** for getting the last part of a CrvData.

subCrvData(*CrvData*, *Start*, *Size*)

Returns	<crvData> A subset of the given curve interpolation data.
CrvData	<crvData> A CrvData.
Start	<integer> An index where the subset is to begin (zero-origin).
Size	<integer> The number of parameters and values to be extracted.

tailCrvData(*CrvData*, *Size*)

Returns	<crvData> A subset from the end of the curve interpolation data.
CrvData	<crvData> A CrvData.
Size	<integer> Number of parameters and values to be extracted.

8.18.2 General Surface Interpolation

The only surface interpolation capability available currently is the general one, which is analogous to the general curve interpolation routine. The construction of the fitting specification is the same as for curves, except that the CrvData will be replaced by a SrfData.

Once a surface fitting specification has been created, a call to **srfInterp** will produce the surface. The **srfInterp** function returns the unique surface (if it exists) that satisfies the fitting specification.

srfInterp(*FitSpec*)

Returns	<surface> A surface which interpolates the given data.
FitSpec	<fitSpec> Interpolation data for a surface.

SrfData

A SrfData contains fitting data for a surface. Its creator is **srfData**.

srfData(*Param*, *Values*)

Returns	<srfData> An object representing the data for interpolation of a surface.
Param	<listOf number | vectorOf number> Ordered list or vector of numbers specifying U parameter values corresponding to the data.
Values	<listOf curve | listOf fitSpec | vectorOf curve | vectorOf fitSpec> List or vector of curves or (curve) FitSpecs, one for each Param. Control points for the curves should all be of the same type.

The following example specifies four circles for surface interpolation by instancing a single circle to establish the curves which will be interpolated:

```
Circle := objTransform( UnitCircle, sg( .5 ))$

Circle1 := objTransform( Circle, rX( 15 ), tZ( .5 ) )$
Circle2 := objTransform( Circle1, rX( 30 ), tZ( 1.0 ) )$
Circle3 := objTransform( Circle2, rX( 45 ), tZ( 1.5) )$
CircleList := list( Circle, Circle1, Circle2, Circle3 )$

PositionSrfData := srfData( '(0 1 2 3), CircleList );
```

A helper function for building srfData objects is **constantCrv**. It returns a curve that always has the specified value. This is useful for specifying constant tangent values at a parametric value.

constantCrv(*Value*)

Returns	<curve> A curve with constant value.
Value	<point | geomVector> The value of the curve.

For example, to specify end tangents for the above surface interpolation:

```
TangentSrfData := srfData( '(0 3),
                           list( constantCrv vec( 0, 0, .5 ),
                                 constantCrv vec( 0, -.5, 0 )));
```

The functions **subSrfData** and **tailSrfData** are analogous to the **subCrvData** and **tailCrvData** functions described in the previous section.

subSrfData(*SrfData*, *Start*, *Size*)

Returns	<srfData> A subset of the given surface interpolation data.
SrfData	<srfData> The surface interpolation data.
Start	<integer> An index where the subset is to begin (zero-origin).
Size	<integer> Number of parameters and values to be extracted.

tailSrfData(*SrfData*, *Size*)

Returns	<srfData> A subset from the end of the surface interpolation data.
CrvData	<srfData> A SrfData.
Size	<integer> Number of parameters and values to be extracted.

8.18.3 Interpolation Examples

Several complete examples of general curve and surface interpolation are provided in this section. The two curve examples can be compared with the results of the corresponding special curve interpolation functions. The two surface examples show interpolation from a set of curve data and from a grid of point data.

Complete Interpolation Example

This example uses the positional data stored in PosData from earlier examples in this section, and goes through the explicit interpolation steps that are used by

```
C2 := CompleteCubicInterp( ChordLength, PosData );
```

The corresponding general interpolation follows:

```
% Generate a chord-length parameter estimation and build CrvData.
Parameters := paramChordLength( PosData );
PositionDataBlock := crvData( Parameters, PosData );

% Create the DataItems for the positional information.
PosDataItem := dataItem( Position, ParamData, ParamLocs,
                         PositionDataBlock )$

% Create DataItems to estimate beginning and end tangents. Use 3
% positions to estimate each tangent.

% Beginning tangent.
BeginTanDataItem := dataItem( Tangent, EstTangent, ParamLocs[0],
                              subCrvData( PositionDataBlock, 0, 3 ));

% End tangent.
EndTanDataItem := dataItem( Tangent, EstTangent,
                            ParamLocs[size ParamLocs],
                            tailCrvData( PositionDataBlock, 3 ));


FitSpec := fitSpec( parmInfo( CUBIC, EC_OPEN, KV_COMPLETE ),
                    PosDataItem, BeginTanDataItem, EndTanDataItem )$


C2 := crvInterp FitSpec;
```

Not-a-Knot Interpolation Example

This example uses the positional data in **PosData** again, and goes through the explicit interpolation steps that are used by

```
C8 := cubicCrvFromParametersAndPositions( ChordLength, PosData );

Parameters := paramChordLength( PosData );
PositionDataBlock := crvData( Parameters, PosData );

PosDataItem := dataItem( Position, ParamData, ParamLocs,
                         PositionDataBlock )$
```

```
FitSpec := fitSpec( parmInfo( CUBIC, EC_OPEN,
                              KV_COMPLETE(
                                KV_Mod_NotAKnot(
                                    KnotLocs[1],
                                    KnotLocs[size(KnotLocs)-1]))),
                    PosDataItem )$
C8 := crvInterp FitSpec;
```

Surface Interpolation Example

The section on surface interpolation described creation of a set of position and tangent information for a srfData (see section 8.18.2 [General Surface Interpolation], page 174), using those examples,

```
SrfFitSpec := fitSpec( parmInfo( CUBIC, EC_OPEN, KV_UNIFORM ),
                       dataItem( position, paramData,
                                 ParamLocs, PositionSrfData ),
                       dataItem( tangent, paramData,
                                 ParamLocs, TangentSrfData ))$

Srf1 := srfInterp SrfFitSpec$
```

Surface Mesh Interpolation Example

A mesh of data points is taken off a sphere of radius 2 and a spline fit through them. Make the spline cubic in the U direction and quadratic in the V direction.

```
NumRows := 5;
NumCols := 6;
DataMesh := mkVect( NumRows - 1 );

for I := 0 : NumRows-1 do

    DataMesh[I] := mkVect( NumCols - 1 );
    for J := 0 : NumCols-1 do

        U := (2.0*J)/(NumCols-1)-1;
        V := (2.0*I)/(NumRows-1)-1;
        DataMesh[I][J] := pt( U, V, sqrt( 4-U*U-V*V ));
    ;
;

% Use uniform parameters.
VParms := numRamp( NumCols, 0, 1 );
UParms := numRamp( NumRows, 0, 1 );

UParminfo := parmInfo( CUBIC, EC_OPEN, KV_UNIFORM );
VParminfo := parmInfo( QUADRATIC, EC_OPEN, KV_UNIFORM );

% Create a vector of curve fitspecs specifying curves in V-direction.
CrvFitSpecs := mkVect(NumRows - 1);

for I := 0 : size CrvFitSpecs do
    CrvFitSpecs[I] := fitSpec(
```

```
                                    VParminfo,
                                    dataItem( Position, ParamData, ParamLocs,
                                              crvData( VParms, DataMesh[I] )));

% Create a surface FitSpec.
FitSpec := fitSpec( UParminfo,
                    dataItem( Position, ParamData, ParamLocs,
                              srfData( UParms, CrvFitSpecs )))$

Srf2 := srfInterp FitSpec;
```

8.19 Animation

IOU - Animation (gm)

8.20 Dimensions Package

The dimensions package is implemented as a set of three parametric types: AngDimAttr, DiaDimAttr, and LinDimAttr. All three provide a **makeGeom** method for generating a geometric description of the dimension. The angular and linear dimension break the geometry construction into three cases, depending on the attribute flags. The text can be inside or outside the dimension; if inside, the arrows can be inside or outside. For diameter dimensions, there are two cases radius and diameter.

dimLinTextInArrInGeom(*Self*)

Returns <group> Construct the geometry to represent a linear dimension with text and arrow inside the arrow.

Self <linDim> The linear dimension.

dimLinTextOutArrOutGeom(*Self*)

Returns <group> Construct the geometry to represent a linear dimension with text and arrow outside the dimension area.

Self <linDim> The linear dimension.

dimLinTextInArrOutGeom(*Self*)

Returns <group> Construct the geometry to represent a linear dimension with text inside and arrow outside the dimension area.

Self <linDim> The linear dimension.

dimAngTextInArrInGeom(*Self*)

Returns <group> Construct the geometry to represent an angular dimension with text and arrow inside the dimension area.

Self <angDim> The angular dimension.

dimAngTextOutArrOutGeom(*Self*)

Returns <group> Construct the geometry to represent an angular dimension with text and arrow outside the dimension area.

Self <angDim> The angular dimension.

dimAngTextInArrOutGeom(*Self*)

Returns <group> Construct the geometry to represent an angular dimension with text inside and arrow outside the dimension area.
Self <angDim> The angular dimension.

dimDiaGeom(*Self*)

Returns <group> Construct the geometry to represent a diameter dimension.
Self <diaDim> The diameter dimension.

dimRadGeom(*Self*)

Returns <group> Construct the geometry to represent a radius dimension.
Self <diaDim> The radius dimension.

A few utility functions are used for the dimensions package. The attributes of dimensions are prefixed by a special keyword (**'Dim**), which effectively makes a "private space" for all the dimension attributes. A set of simple wrapper functions to the standard attribute package is used.

setDimAttr((*Obj*, *Name*, *Value*))

Returns <Nil> Set an attribute on a dimension object.
Object <diaDim | linDim | angDim> The object to set the attribute for.
Name <keyword> The name of the attribute.
Value <anything> The value of the attribute.

getBoundDimAttr(*Dimension*, *AttrName*)

Returns <any | '!*Unbound!*> Get a dimension attribute from a dimenion object, returning the unbound token if the attribute is not present.
Dimension <linDim | diaDim | angDim> The dimension object to get the attribute from.
AttrName <keyword> The name of the attribute to retrieve.

dimAttrP((*Object*, *Name*))

Returns <boolean> See if an attribute is bound for a dimension.
Object <linDim | angDim | diaDim> The object to check.
Name <keyword> The name of the attribute to look for.

addDimAttrs(*Object*, *OptionsPile*)

Returns <linDim | angDim | diaDim> The object to which the attributes were attached.
Object <linDim | angDim | diaDim> The object to attach the attributes to.
OptionsPile <list> A number of dimension attributes which are to be added to the object.

The dimensions package also uses a set of arrow creation functions.

setArrowHeadLength(*Len*)

Returns <Nil> Set the global variable controlling length of arrowheads.
Len <number> The new length value.

PtArrowHeadGeom(*Point*, *Delta*)

Returns <listOf point> Construct geometry to represent the arrowhead.
Point <point> Location of tip.
Delta <geomVector> The direction of the arrow.

PtArrowGeom(*Point0*, *Point1*)

Returns <listOf point> Construct geometry to represent the arrow.
Point0, Point1
<point> End points of the arrow, head is at *Point1*.

ArrowGeom(*Point0*, *Point1*)

Returns <polyline> Produce a polyline to represent the arrow geometry.
Point0, Point1
<point> End points of the arrow, head is at *Point1*.

ArrowHeadGeom(*Point*, *Direction*)

Returns <polyline> Produce a polyline to represent the arrowhead geometry.
Point <point> Location of tip of arrowhead.
Direction <geomVector> Direction of the arrowhead.

BentArrowGeom(*Tail*, *Middle*, *Tip*)

Returns <polyline> Produce a polyline to represent bent arrow geometry.
Tail, Middle, Tip
<point> The defining points for the bent arrow.

CurvedArrowGeom(*End0*, *Center*, *End1*)

Returns <group> An arc and polyline representing the curved arrow geometry.
End0, End1
<point> End points of the arc, *End1* is at the arrowhead.
Center <point> Center of the arc.

CurvedBentArrowGeom(*Tail*, *End0*, *Center*, *End1*)

Returns <group> A group with an arc for the curved part of the arrow, and polylines for the head and straight tail.
Tail, End0, Center, End1
<point> Defining points for the arrow.

CurvedDoubleArrowGeom(*End0*, *Center*, *End1*)

Returns <group> A group containing the arc for the curved arrow and polylines for an arrowhead at each end.
End0, Center, End1
<point> Defining points for the arrow.

8.21 DumpA1

The **dumpA1File** operation is based on **dumpA1** methods provided by all the objects. After opening the output file, **dumpA1File** calls **dumpA1Obj** which breaks up lisp vectors and lists until it gets to objects. Then it calls the **dumpA1** method for the object. Most of the **dumpA1** methods use **dumpIndent**, **printHeader**, and **dumpA1ObjName** as standard building blocks. Both curves

and surface use **dumpA1MeshPts** and polygons and polylines use **dumpA1PolyPts**. For adjacency declarations, **edgeNum** translates symbolic edge references to numeric codes understood by the C structures.

dumpA1Obj(*ObjStruct*, *OutChnl*)

Returns	<Nil> Dumps the objects given into the output file in Alpha_1 text file format. (Must be called by **dumpA1File**).
ObjStruct	<anything> The stuff to be dumped.
OutChnl	<integer> Output channel.

dumpal(*Self*, *OutChnl*, *Level*)

Returns	<Nil> Dumps the object to the output file.
Self	<object> The object to be dumped.
OutChnl	<integer> Current output channel.
Level	<integer> Current indentation level.

dumpIndent(*Level*, *OutChnl*)

Returns	<Nil> Writes tabs and spaces to output file.
Level	<integer> How many tabstop levels to indent.
OutChnl	<integer> Current output channel.

printHeader(*objType*, *obj*, *OutChnl*)

Returns	<Nil> Print header information for an object to the output file.
objType	<string> Name of the object, to be printed in the text file.
obj	<object> The object being dumped.
OutChnl	<integer> Current output channel.

dumpA1ObjName(*Name*, *OutChnl*, *Level*)

Returns	<Nil> Dumps name of the object as a string attribute (if not Nil).
Name	<string> The name of the object.
OutChnl	<integer> Current output channel.
Level	<integer> Current indentation level.

dumpA1MeshPts(*Mesh*, *OutChnl*, *Level*)

Returns	<Nil> Dump a control mesh to the file.
Mesh	<ctlMesh> The control mesh to dump.
OutChnl	<integer> Current output channel.
Level	<integer> Current indentation level.

dumpA1PolyPts(*Poly*, *OutChnl*, *Level*)

Returns	<Nil> Dump a control polygon to the file.
Poly	<ctlPolygon> The control polygon to dump.
OutChnl	<integer> Current output channel.
Level	<integer> Current indentation level.

edgeNum(*Edge*)

Returns	<integer> Turn an edge keyword into a numeric value for text file.
Edge	<keyword> The edge keyword, e.g. **'TOP**.

dumpboolean((*bool*))

Returns	<0 | 1> Convert a boolean symbol into a numeric value for text file.
bool	<boolean> The value to convert.

8.22 ChannelPrinters

The **channelPrin** methods are the default printing form used by **shape_edit** (PSL really) for all objects. It is thus important to make sure that a reasonable **channelPrin** method is provided, because you're going to be seeing it a lot. Most of the time you should not need to define one: a default is provided by **defObject**, and **defModelObject** overrides it with a default better suited to Alpha_1. If these are not acceptable, your best bet is to copy an existing one and modify it. The form of the procedure is important for getting correct indentation in the output, and is dictated by the underlying lisp system. The description in this section doesn't attempt to explain why the **channelPrin** methods look the way they do or how they work, but should give you the information necessary to build **channelPrin** methods for new objects if necessary.

The **objChannelPrint** function should be called whenever a **channelPrin** method wants to print the value of a slot which is expected to contain another object. The function **channelPrinIndentLevel** tabs over to the indentation level specified and should be called before each **channelPrintf** line.

objChannelPrint(*ObjArg*, *Channel*, *IndentLevel*, *Print1!?*)

Returns	<Nil> Calls the **channelPrin** methods for objects.
ObjArg	<object> The object to be printed.
Channel	<integer> Current output channel.
IndentLevel	<integer> Current indentation level.
Print1!	<boolean> Not used.

channelPrin(*Self*, *Channel*, *Level*, *Print1!?*)

Returns	<Nil> Prints the object to the output channel.
Self	<object> The object to be printed.
Channel	<integer> Current output channel.
Level	<integer> Current indentation level.
Print1!	<boolean> Not used.

channelPrinIndentLevel(*Channel*, *IndentLevel*)

Returns	<Nil> Indent to given level on current channel.
Channel	<integer> Current output channel.
IndentLevel	<integer> Current indentation level.

The **IndentBase!*** variable is the "magic key" to getting the indentation where you want it, and is the reason that **channelPrin** methods are so complex. Before calling **objChannelPrint** to print sub-objects of an object, you must rebind the **IndentBase!*** (if you want nested indentation of the sub-objects). An *expression* involving **IndentBase!*** is acceptable as an argument to **channelPrinIndentLevel**, but will not work as an argument to **objChannelPrint**. The preferred formatting style is something like:

```
#<NewObject
    field1 = 2
```

```
    field2 =
        #<SubObject ...
        >
>
```

Since the call to **objChannelPrint** for the sub-object is two indentation levels below the original (because the name of the field takes one level), the **IndentBase!*** is most easily handled by rebinding it as:

```
IndentBase!* := IndentBase!* + 2;
```

and using

```
channelPrinIndentLevel( Channel, IndentBase!* - 1 );
```

for indenting before printing the name of the field. Don't forget to rebind **IndentBase!*** before printing the closing brace of the original object.

Other useful utilities are **tabStopLevel** and **fieldFitOnLine**. All of the single parameter matrix descriptor objects have **channelPrin** methods which **callprTransObj1Param**. Objects based on **floatArrays** are printed with **prettyPrintFloatArray**, although **printFloatArray** is faster.

tabStopLevel(*IndentLevel*)

Returns <Nil> Sets the tabstop level for cannel printing.
IndentLevel
<integer> New indentation level.

fieldFitOnLine(*FieldPosn*, *FieldSize*)

Returns <boolean> Whether the given field will fit on the line.
FieldPosn <integer> Position where the field will start.
FieldSize <integer> Size of the field.

prTransObj1Param(*Obj*, *FieldName*, *Channel*, *Level*, *Print1!?*)

Returns <Nil> Print a matrix descriptor object with one parameter.
Obj <object> The matrix descriptor.
FieldName
<string> Name to use as the field name.
Channel <integer> Current output channel.
Level <integer> Current indentation level.
Print1! <boolean> Not used.

printFloatArray(*FArray*, *Channel*, *IndentLevel*, *Print1!?*)

Returns <Nil> Print a float array object.
FArray <floatArray> The packed array of floats.
Channel <integer> Current output channel.
Level <integer> Current indentation level.
Print1! <boolean> Not used.

prettyPrintFloatArray(*FArray*, *Channel*, *IndentLevel*, *Print1!?*)

Returns <Nil> Prints a pretty form of a packed array of floats.
FArray <floatArray> Packed array of floats.
Channel <integer> Current output channel.

Level	<integer> Current indentation level.
Print1!	<boolean> Not used.

8.23 Save & Restore

The **shape_edit** functions for saving and restoring models are based on two facilities implemented with a small set of routines. The first part is methods for objects which produce code that, when evaluated, will rebuild the object with appropriate values in all its slots. The other part is routines which trace through the model dependency graph, deciding which objects to save, writing out the dependency information, and relinking the grah when the model is restored. The file which is produced for a saved model is actually just a standard Rlisp source file. When interpreted, that code restores the model to the original state at the time of the save. For very small models, rebuilding from the original construction statements may be nearly as fast. For large models, the saved form should be must faster since the actual geometric data is stored in the file and need not be recalculated.

The **consform** methods for each object generate code which rebuilds the object. the **consform** methods are generated for individual object by **defModelObject** when the objects are defined. The **consformGeneral** function is a wrapper routine which handles lists and vectors and calls the object methods when appropriate. I tallows code for regenerating parts of the model to correctly handle embedded list and vector objects. The **consformFloatArray** function is used to handle packed floating point array types, since these are not built in the objects package, but a t a lower level. The **rlispPrintConsform** function prints the code produced by the consform methods in Rlisp syntax, with a reasonable indentation style.

consform(*Self*)

Returns	<list> A lisp expression which, when evaluated, will rebuild a copy of the object.
Self	<object> The object for which the code is to be generated.

consFormGeneral(*Thing*)

Returns	<list> A lisp expression to reconstruct general structures which include lists and vectors as well as objects.
Thing	<object | listOf object | vectorOf object> The structure for which code is to be generated.

consFormFloatArray(*Thing*)

Returns	<list> A lisp expression which will rebuild the packed array of floats when evaluated.
Thing	<floatArray> The array for which the code is to be generated.

rlispPrintConsForm(*ConsForm*)

Returns	<Nil> Prints an expression in Rlisp syntax (to standard output), optimized to look good for commonly stored Alpha_1 objects like surfaces and curves.
ConsForm	<list> The lisp expression to be printed, usually generated by the **consform** methods.

rlispPrintConsForm2(*ConsForm*, *OutChnl*, *Level*)

Returns	<Nil> Prints an expression to the given channel in Rlsip syntax.

ConsForm	<list> The lisp expression to be printed, usually generated by the **consform** methods.
OutChnl	<integer> Current output channel.
Level	<integer> Current indentation level.

The functions which search through the model dependency graph to find objects which need to be saved uses the **ancestorCount** and **mark** fields of model objects. The **clearCnt** function clears these fields at the beginning and end of the save process. There are two phases to saving a model: the marking and collection phase, and the linking phase. In order for the restore functions to be able to link the model graph back up, the objects must be written in an order such that no object is written out before any object upon which it depends. For example, consider an arc which is constructed tangent to three lines, each of which is constructed based on two points. The arc must be written after the lines, which in turn must be written after the points. In the first phase, countPrereqs walks through the model graph marking objects which need to be output and collecting a list of them in the order in which they are encountered. This list of objects is then written out in reverse order, with the **consform** methods being used to generate the code to rebuild the objects. The **doLinking** and **processPrereqs** functions generate the code which will restore all the model dependency links when executed. In order to be able to reference other objects, but not have to name them, the save routines assign a position in a global vector called **objRefs!*** to all objects which need references. The **processPrereqs** function examines a list of prerequisites for an object and inserts references to the object reference vector wherever appropriate.

clearCnt(*Obj*)

Returns	<Nil> Clear all the counts by tracing through the dependency slots.
Obj	<object> The object for which counts are to be cleared.

countPrereqs(*ObjStruct*)

Returns	<listOf object> Trace through prerequisites of an object, marking objects as they are encountered, incrementing the counts, and collecting a list of the objects found.
ObjStruct	<object | listOf object | vectorOf object> The object structure to be traced.

countPrereqs2(*Obj*)

Returns	<listOf object> Does the actual work for **countPrereqs** after lists and vectors are destructured.
Obj	<object> The object to be traced.

doLinking(*ObjStruct*, *OutChnl*)

Returns	<Nil> Generate statements in the output file which rebuild the connections between objects in the model being saved.
ObjStruct	<object | listOf object | vectorOf object> The structure for which linking is to be generated.
OutChnl	<integer> Current output channel.

doLinking2(*Obj*, *OutChnl*)

Returns	<Nil> Does the actual work for **doLinking**, after lists and vectors have been destructured.
Obj	<object> The object for which connection information is to be generated.
OutChnl	<integer> Current output channel.

processPrereqs(*PreList*, *OutChnl*)

Returns	<Nil> Prints prerequisite information about the object to the channel, after doing some processing to resolve references.
PreList	<listOf object> The list of prerequisites.
OutChnl	<integer> Current output channel.

processPrereqs2(*P*, *OutChnl*)

Returns	<Nil> Does the work for **processPrereqs**, after lists and vectors are destructured.
P	<object> The prerequisite object to be output.
OutChnl	<integer> Current output channel.

The code which is generated by **saveModel** uses calls to **rebuildObjConnections** to link the model dependencies back during restore operations. it is quite similar to the general **addToModel** function but doesn't actually build the object (since that's been done earlier in the restoration). The **bindModelObjsToSymtab2** and **bindModelToSymtab2** functions are helpers for the corresponding user level functions.

rebuildObjConnections(*Obj*, *Expr*, *PrereqList*)

Returns	<Nil> Recreate all the dependency links for an object.
Obj	<object> The object to link back into the model.
Expr	<list> The construction expression
PrereqList	<listOf object> The list of prerequisites for the object.

bindModelToSymtab2(*Refs*, *Names*)

Returns	<Nil> Rebind all the objects in a reference vector to names in the symbol table.
Refs	<vectorOf object> The list of objects to bind back into the symbol table.
Names	<vectorOf keyword> Identifiers for all the objects.

bindModelObjsToSymtab2(*Refs*, *Names*, *IndexList*)

Returns	<Nil> Rebind specified objects in a reference vector to names in the symbol table.
Refs	<vectorOf object> The list of objects to bind back into the symbol table.
Names	<vectorOfOf keyword> Identifiers for all the objects.
IndexList	<listOf integer> The list of indices for the objects which are to be bound.

showRestoredModel2(*ObjRefs*)

Returns	<Nil> Display the objects in the object reference vector.
ObjRefs	<vectorOf object> The object reference vector.

During restore operations, objects based on packed arrays of floating point numbers use **mkFloatArray** to reconstruct the contents of the array slot. It is not an amenable form for ever writing by hand.

mkFloatArray(*RowSz*, *ColSz*, *PtType*, *NCoords*, *Vals*)

Returns	<floatArray> Reconstruct a packed array of floats from a list of values.
RowSz, *ColSz*	<integer> Dimensions of the array.

PtType	<keyword> Base type of the elements.
NCoords	<integer> Number of coordinates in the base element.
Vals	<listOf float> The list of float values to put in the array.

8.24 Display Package

A large number of functions are described in the User's Manual for handling the display of models and the windows and screens associated wiht display devices. Most of those functions are simple macro wrappers which make the oeprations more palatable for users. In particular, the references to window and devices do not require quoting from the user level, although the identifiers must be properly quoted by the time they are evaluated. So every user level function involving device or window names is a macro which quotes the names and calls the companion routine which has the suffix "byName". For completeness, these functions are liested below, but no detailed descriptions of the functions or their arguments are provided, since the parallels are so obvious.

grabDeviceByName(*DeviceName*)

dropDeviceByName(*DeviceName*)

switchToDeviceByName(*DeviceName*)

flushDeviceByName(*DeviceName*)

clearDeviceByName(*DeviceName*)

newWindowByName(*WindowName*)

newDevWindowByName(*DeviceName*, *WindowName*)

selectWindowByName(*WindowName*)

selectDevWindowByName(*DeviceName*, *WindowName*)

deselectWindowByName(*WindowName*)

deselectDevWindowByName(*DeviceName*, *WindowName*)

copyDevWindowByName(*DeviceName*, *WindowName*)

moveDevWindowByName(*DeviceName*, *WindowName*)

removeWindowByName(*WindowName*)

removeDevWindowByName(*DeviceName*, *WindowName*)

clearWindowByName(*WindowName*)

clearDevWindowByName(*DeviceName*, *WindowName*)

setWindowViewmatByName(*WindowName*, *ViewMat*)

setDevWindowViewmatByName(*DeviceName*, *WindowName*, *ViewMat*)

objectsInDevWindowByName(*DeviceName*, *WindowName*)

objectsInWindowByName(*WindowName*)

The **show**, **unshow**, and **reshow** functions take a variable number of argumetns which are formed into a list and passed on to companion functions as well. The **highlight** and **unhighlight** functions are similar.

showList(*ObjectList*)

unshowList(*ObjectList*)

reshowList(*ObjectList*)

displayObject(*Object*)

undisplayObject(*Object*)

showInWindowList(*WindowName, ObjectList*)

unshowFromWindowList(*WindowName, ObjectList*)

showObjectInDevWindow(*Obj, DevName, WindowName*)

removeObjectFromDevWindow(*Object, DeviceName, WindowName*)

highLightList(*ObjList*)

unHighLightList(*ObjList*)

highlightDevObject(*DeviceName, Object*)

unhighlightDevObject(*DeviceName, Object*)

unhighlightObjectFromDevWindow(*Obj, DevName, WindowName*)

redisplayObject(*Object*)

A routine which actually displays a particular type of object may be one of several types. Objects which have C data structure equivalents and Lisp routines for passing the data across (**toC** methods) are sent directly to the device handler process. Curves and surfaces are two important object types which fall into this class. Some objectsw have specialized C data structure equivalents which are used only for display purposes and have **dlsToC** methods for transferring the data directly to the device handler process. Groups and instances fall into this class. All objects provide a **dlsObj** method which is actually used for display, but the ones for objects which have **toC** or **dlsToC** methods are generated automatically by the system with a **defineDlsObjMethods** statement. Other objects must be converted to a geometric description based on objects in these classes (which have C equivalents) in order to be displayed. These conversions are performed in a **dlsObj** method for the object which will call **dlsObj** on the derived geometric representation. Examples in this class are arcs (which are converted to spline curves for display) and polygons (which are converted to polylines for display). The **dlsObjCommon** routine is used at the bottom level for display of all objects. It invokes the C code which sends the objects through the binary stream to the device handler process. Objects which are aggregates (groups, instances, shells, and combiner objects) must also provide a **redisplay** method to propagate **reshow** commands to the component objects. The **dlsLine** is a common routine that is shared by the **dlsObj** methods for both 2D and 3D lines.

dlsObj(*Self, Seg*)

Returns	<Nil> Displays the object by calling **dlsObjCommon** or another **dlsObj** method.
Self	<object> The object to be displayed.
Seg	<integer | Nil> Device segment handle.

dlsObjCommon(*Object, Seg, FreeTempSpace, C_Temp_Ptr*)

Returns	<Nil> Display the object, calling C code.
Object	<object> The object to be displayed.
Seg	<integer | Nil> Device segment handle.
FreeTempSpace	<boolean> Whether or not to free the temporary malloc storage.

C_Temp_Ptr
<cPointer> Pointer to C equivalent of the object.

redisplay(*Self*, *DeviceName*)

Returns <Nil> Redraw the object.
Self <object> The object to redraw.
DeviceName
<keyword> Name of the device on which to redraw the object.

dlsLine(*Ln*, *Seg*)

Returns <Nil> Display an infinite line by turning it into a finite polyline.
Ln <line> The line.
Seg <integer> Device segment handle.

Display parameters for objects which are passed directly to the C device handler process must be controlled by setting variables in that process. Some of these parameter setting functions are described in the User's Manual (e.g. **setSrfFineness**). Some more esoteric ones are listed below. These are not intended for users at all (even advanced ones), but are provided as a mechanism for fine-tuning the display algorithms. The minimum refinement value sets a minimum number of knots which will be added in one polynomial span of a curve for display. Functions are available for setting a bounding box which is used in computing the scaling, and for setting the automatic scaling value. The device address is the name of the host on which the device handler is to be found, and the device command specifies the name of the device handler program. The device arguments which are passed to the device handler when it is started can also be set. These commands are useful when building or modifying display device handler programs.

getMinRefinement()

setMinRefinement(*Int*)

getXmin()

setXmin(*Number*)

getYmin()

setYmin(*Number*)

getZmin()

setZmin(*Number*)

getXmax()

setXmax(*Number*)

getYmax()

setYmax(*Number*)

getZmax()

setZmax(*Number*)

getAutoscale()

setAutoscale(*Number*)

deviceAddress(*DeviceName*)

setDeviceAddress(*DeviceName*, *Value*)

deviceCmd(*DeviceName*)

setDeviceCmd(*DeviceName*, *Value*)

deviceArgs(*DeviceName*)

setDeviceArgs(*DeviceName*, *Value*)

A few other functions which are used in the display package are described below.

showDevText(*DeviceName*, *Text*)

Returns <Nil> Display the text on the named device.
DeviceName
<keyword> Name of the device.
Text <textObject> The text string and location.

deviceFromName(*DeviceName*)

Returns <deviceObject> The device state structure for the given device.
DeviceName
<keyword> The name of the device.

deviceWindow(*Device*, *WindowName*, *ReportError*)

Returns <windowObject> The window associated with a device.
Device <deviceObject> The device state structure.
WindowName
<keyword> The name of the window.
ReportError
<boolean> Whether or not to report errors.

segmentOnDevice(*Object*, *DeviceName*, *ReportError*)

Returns <integer> The device segment handle for the given object.
Object <object> The object to look for.
DeviceName
<deviceObject> The device state structure.
ReportError
<boolean> Whether to report errors.

!&cSelectDevice(*DeviceName*)

Returns <boolean> Select a device as the current one. Returns Nil if it did not succeed.
DeviceName
<deviceObject> The device state structure.

maybeCreateSegmentForObject(*DeviceName*, *Object*)

Returns <integer> Device segment handle, perhaps newly created.
DeviceName
<keyword> The name of the device.
Object <object> The object to be displayed.

addObjectToDevice(*Dev*, *Object*, *CSeg*)

Returns <Nil> Link the object into the device state structure.

Dev <deviceObject> The device state structure.
Object <object> The object to be displayed.
CSeg <integer> The device segment handle.

removeObjectFromDevice(*Obj*, *DeviceName*)

Returns <Nil> Unlink the object (remove it from the display) from the device state structure.
Obj <object> The object to be removed.
DeviceName
<keyword> Name of the device.

objectFromSeg(*Dev*, *CSeg*)

Returns <object> The object referred to by a given device segment handle.
Dev <deviceObject> The device state structure.
CSeg <integer> Device segment handle.

dlSegHash(*CSeg*)

Returns <Nil> Hash device segment handles into slots of the device state structure.
CSeg <integer> Device segment handle.

modifySegmentForObject(*DeviceName*, *Object*)

Returns <Nil> Display a new version of an object.
DeviceName
<keyword> Name of the display device.
Object <object> The object to be modified.

selectedWindowP(*WindowName*)

Returns <boolean> Whether the window is selected on the current device.
WindowName
<keyword> Name of the window.

selectedDevWindowP(*DeviceName*, *WindowName*)

Returns <boolean> Whether the window is selected on the specified device.
DeviceName
<keyword> Name of the device.
WindowName
<keyword> Name of the window.

copyDevWindowToFrom(*NewDevName*, *OldDevName*, *WindowName*)

Returns <Nil> Copy the contents of one window to another device.
NewDevName
<keyword> Name of the new device.
OldDevName
<keyword> Name of the old device.
WindowName
<keyword> Name of the window.

8.25 GUI General Package

This section (and the next several) describe internal functions used in the graphical (menu-driven) user interface to **shape_edit**. Of course, by far the majority of functions used in the GUI are just the usual functions provided in the command-driven programmer's interface.

IOU - A short summary of the philosophy and design of the GUI. (esc)

The remainder of this section describes the control functions which actually are running the graphical interaction. The next section describes some functions which are used to provide a smoother transition from functions invoked on the menus to the programming invocation of the functions. Finally, there is a section which describes functions which get arguments for routines invoked from the graphical user interface.

The **interact** procedure which is invoked from the user level calls **interactWithDevice**, which contains the real interaction loop. This loop waits for events to occur on the display, taking an appropriate action. In most cases, that action is to call **handleEvent** which contains a case statement for each of the event types (menu, locator, etc.). At the moment, locator events in the GUI are always immediately converted to points (with **ptFromLocatorEvent**) and displayed. Occasionally, it is useful to have separate interactive modules within the GUI. For this purpose, the function **subInteract** can be called. It allows specification of the menu and level from which to continue interaction. The **showPrompt** routine is always used to display prompts (although often the prompt strings are constructed with **stdPrompt**, which then calls **showPrompt**), and the **showMenu** routine is always used to display menus. The interaction loop normally passes objects back and forth during processing (the objects being constructed by the user), but there are also a number of control tokens which are used for special commands affecting the interaction (e.g., aborting a construction). These control tokens are identified using **controlTokenP**.

interactWithDevice(*DeviceName*, *MenuName*, *LevelName*)

Returns <Nil> Initiates interaction with the specified device.
DeviceName
<keyword> Name of the device.
MenuName
<string> Name of the menu.
LevelName
<string> Description of current level.

subInteract(*MenuName*, *LevelName*)

Returns <Nil> Initiates a sub-interaction loop within the main interaction.
MenuName
<string> The name of the menu.
LevelName
<string> Description of current level.

handleEvent(*DeviceName*)

Returns <anything> The result of handling an event.
DeviceName
<keyword> Name of the device.

showPrompt(*Prompt*)

Returns <Nil> Display a prompt line on the interaction device.

Prompt <textObject> The prompt and its location.

stdPrompt(*ArgDescription*)

Returns **<Nil> Display a standard prompt (used by the "get" functions).**
ArgDescription
<string> Description of desired argument.

showMenu(*MenuName*)

Returns <Nil> Set the current menu on the interaction device.
MenuName
<string> Name of the menu.

controlTokenP(*Item*)

Returns <boolean> Whether an item is a control token for the interaction.
Item <anything> The item to test.

ptFromLocatorEvent(*Loc*)

Returns <point> A point constructed from a locator event.
Loc <locatorEvent> A locator event from the interaction device.

A few special functions which are initiated by keyboard events rather than menu selections are **repeatLastAction** (for redoing a command), **showInteractionCmdStack** (for tracebacks), and **showInteractiveHelp** (for help).

repeatLastAction()

Returns <Nil> Perform the last selected menu action again.

showInteractionCmdStack()

Returns <'!$NoOp!$> Displays the nested commands being executed in the prompt window.

showInteractiveHelp()

Returns <'!$NoOp!$> Displays help strings in the prompt window.

Most selections from the menus initiate a sequence of actions which are aimed at providing all the arguments for a particular function and then executing that function with the arguments which were collected. The **promptForArgsAndCall** function is a sort of interpreter of the menu item data structure, which generates appropriate prompt strings and collects arguments until the desired function can be executed. In addition to creating objects, the GUI also allows the construction of existing objects to be modified. This is accomplished with **editConstruction** which retrieves the information about how the object was constructed in the first place, and then re-enters the construction phase as if the user had just selected all the original parameters but not yet accepted the construction. Both the initial construction (**promptForArgsAndCall**) and editing of a construction (**editConstruction** use a common routine called **constructInteractively** which does the actual work of generating prompts, gathering arguments, and executing the construction. Whenever a variable number of arguments is allowed in a function, the GUI uses **getList** to gather the list of arguments. During editing, **constructInteractively** uses **editList** to update an existing argument list. The editList function is also used by **getList**. When gathering arguments, the various "get" functions use **setPickTypes** restrict the type of objects that can be picked from the display to fulfill a particular argument value.

promptForArgsAndCall(*FnToCall*, *ArgExprs*)

Returns <anything> Prompt for the arguments to a function and evaluate it when they are all present.
FnToCall <keyword> Name of the function to call.
ArgExprs <list> List of expressions to evaluate to generate prompts and get arguments.

editConstruction(*Obj*)

Returns <anything> Re-enter editing the construction of a model object.
Obj <object> The object to edit.

constructInteractively(*Constructor*, *ArgExprs*, *Obj*)

Returns <anything> Do interactive construction of an object.
Constructor <keyword> Name of the function to call.
ArgExprs <list> List of expressions to evaluate to generate prompts and get arguments.
Obj <object> The object to modify (if editing), Nil otherwise.

getList(*GetFn*, *MaxNbr*, *PromptStrings*)

Returns <list> Collect a list of values interactively as an argument for a function.
GetFn <keyword> Name of the function to use to get individual elements of the list.
MaxNbr <integer> A maximum length for the list (Nil if there is no limit).
PromptStrings <string> The prompt strings to use. The last one is repeated if necessary.

setPickTypes(*PredicateFns*)

Returns <Nil> Set the pickable types.
PredicateFns <listOf keywords> List of functions to be applied to see if an object is one of the desired types.

Since display devices all have different keyboards and button configurations, the functions **buttonNameFromNumber** and **buttonNumberFromName** are used in the GUI to provide a device independent layer for identifying button events in **shape_edit**.

buttonNameFromNumber(*Nbr*)

Returns <keyword> Convert a button number to a keyword name.
Nbr <integer> The button number.

buttonNumberFromName(*Name*)

Returns <integer> Convert a keyword button name into a button number.
Name <keyword> The buttom name.

8.26 GUI Menus

The menus are, of course, a major part of the GUI. There is just one simple function for creating the

menu data structures. All menus are created by a call to **addMenuItem** for each item which is to appear in the menu. New sub-menus are added at the end of the existing sub-menus, and new items and the end of the existing items. Blank separator lines in the menus are generated with menu items that have null title strings and actions. The helper functions, **findMenu**, **menuNameFromNumber**, **menuNumberFromName**, **itemNumberFromName**, and **itemNameFromNumber**, are used by **addMenuItem** as well as by other routines which need to access the menus.

addMenuItem(*Menus*, *MTitle*, *ITitle*, *Action*)

Returns <Nil> Add a new menu item to the end of the menu.
Menus <list> The list of menus.
MenuTitle <keyword> Name of the menu in which the item is to be added.
ItemTitleOrPair
<string | pairOf keyword> Title of the item. If it is a pair, replace the existing one.
SubMenuOrAction
<keyword | list> The name of a submenu reached from this item, or the action to be taken when it is selected.

findMenu(*Menus*, *MenuTitle*)

Returns <list> Locate a particular menu in a menu structure.
Menus <list> The menu structure.
MenuTitle <keyword> The title of the menu.

menuNameFromNumber(*Nbr*)

Returns <keyword> Look up the name of a menu in the menus structure.
Nbr <integer> The index of the menu item.

menuNumberFromName(*TitleString*)

Returns <integer> Index of a menu in the menus structure.
TitleString
<keyword> The title of the menu.

itemNumberFromName(*TitleString*, *ItemString*)

Returns <integer> Number of the item.
TitleString
<keyword> Name of the menu where the item is.
ItemString
<keyword> Name of the item.

itemNameFromNumber(*Nbr*, *INbr*)

Returns <keyword> Name of the item.
Nbr <integer> Number of the menu where the item is.
INbr <integer> Number of the item in the menu.

The remainder of this section will describe routines that are loosely associated with particular menus. They are usually wrappers for user level functions which are commonly used in the programming interface, but which are inappropriate for the GUI. There are a variety of reasons for this. A number of functions are related to keywords, which are easy to generate in the programmer's interface since typing is the main communication mode, but which would be cumbersome if

provided that way in the GUI. Some condense classes of functions into a single function in cases where the menus would be unwieldy if every function was entered as a separate menu item.

Predicate functions are provided for classes of keywords. These allow the GUI "get" functions to insure that an appropriate value has been specified for an argument. Luckily, defining a new "class" or argument type for the GUI just involves providing a predicate function which can decide whether any given object is a member of the class and a "get" function which (indirectly) invokes the predicate function. All of these functions take any object as the argument and return a boolean value, so the complete argument descriptions are omitted here.

orderKeywordP(*Val*)

endCondKeywordP(*Val*)

kvKeywordP(*Val*)

axisKeywordP(*Val*)

booleanKeywordP(*Val*)

directionKeywordP(*Val*)

srfEdgeKeywordP(*Val*)

rboxKeywordP(*Val*)

rboxFaceKeywordP(*Val*)

The function **setObjectName** is the mechanism used by the GUI to duplicate the semantics of the Rlisp assignment statement.

setObjectName(*Obj*, *NameStr*)

Returns	<'**!$NoOp$**> Set the name of an object.
Obj	<object> The object.
NameStr	<string> The name.

The numbers menu has a fairly large set of helper functions. Rather than putting coordinate accessors on the menus for both points and vectors, the functions **xCoord**, **yCoord**, and **zCoord** examine the type of the argument and dispatch to the correct accessor. Similarly, **angleBetween** finds to angle between two planes or two lines, and **getRadius** accepts either circles or arcs. The negate function implements unary minus. The **stringFromNumber** function is used by **addConstant** to allow the user to easily access his favorite numerical constants by placing them on a menu.

xCoord(*PtOrVec*)

Returns	<number> X coordinate of a point or vector.
PtOrVec	<point | geomVector> The object to extract the coordinate from.

yCoord(*PtOrVec*)

Returns	<number> Y coordinate of a point or vector.
PtOrVec	<point | geomVector> The object to extract the coordinate from.

zCoord(*PtOrVec*)

Returns	<number> Z coordinate of a point or vector.
PtOrVec	<point | geomVector> The object to extract the coordinate from.

angleBetween(*Obj1*, *Obj2*)

Returns	<number> Angle between two lines or two planes.

Obj1, Obj2
<line | plane> Object to compute angle from (must be of the same type).

getRadius(*ArcOrCir*)

Returns <number> Radius of a circle or arc.
ArcOrCir <arc | circle> Object from which to extract the radius.

negate(*Num*)

Returns <number> Negate a number.
Num <number> The number to negate.

addConstant(*Num*)

Returns <number> Add a constant to the available spaces in the number menu.
Num <number> The constant value to add.

stringFromNumber(*Num*)

Returns <string> A string representation of a number.
Num <number> The number to convert.

On the points menu, the **centerOf** function accepts circles or arcs, and **calcBboxMin** and **calcBbox-Max** compute one of the two bounding box points.

centerOf(*ArcOrCircle*)

Returns <point> The center of an arc or circle.
ArcOrCircle
<arc | circle> Object to extract the center point from.

calcBboxMin(*Obj*)

Returns <point> Minimum point on the bounding box of an object.
Obj <object> The object for which the bounding box is to be calculated.

calcBboxMax(*Obj*)

Returns <point> Maximum point on the bounding box of an object.
Obj <object> The object for which the bounding box is to be calculated.

In order for rounded boxes to be displayed properly when the edge rounding and face opening attributes are added, **setRboxRadiusAndShow** and **setRboxFacesOpenAndShow** are used from the GUI. In order to generate keywords like 'ULedge without having to list them all on a menu, the function **makeRboxEdgeKeyword** is used to construct the edge keyword from two face keywords.

setRboxRadiusAndShow(*Rbox*, *Edge*, *RadiusValue*)

Returns <Nil> Set a radius attribute on a rounded box and redisplay.
Rbox <rBox> The rounded box.
Edge <keyword> Which edge to round, specified as for **setRboxRadius**.
RadiusValue
<number> Radius value for the edge.

setRboxFacesOpenAndShow(*Rbox*, *FaceList*)

Returns <Nil> Set an open face attribute on a rounded box and redisplay.
Rbox <rBox> The rounded box.

FaceList <listOf keyword> List of the faces to declare as open.

makeRboxEdgeKeyword(*Face1*, *Face2*)

Returns <keyword> Construct an edge keyword from two face keywords.

Face1, Face2

<keyword> The face keywords.

Several of the shape operations accept functions from the programmer's interface to describe the shape of the deformation. Since this would be very difficult from the graphical interface, the GUI allows those shape operations to be used by specifying 2D spline curves which are used as descriptions of the function. Although any curve can be used, there are some restrictions on the relation of the knots to the control points (X coordinates of the curves must occur at node values, and hence must also be in increasing sequence). If the curve used does not meet these requirements, then the results may be unintuitive (at best). Four functions are provided for building these "spline function" curves in such a way that the requirements are met: **constantSplineFn**, **linearSplineFn**, **approxValuesSplineFn**, and **interpValuesSplineFn**. Once the curve is defined, it is passed to the shape operator as an extra argument and the procedure **evalSplineFn** is specified for the function argument.

evalSplineFn(*Param*, *Crv*, *OutOfBoundsVal*)

Returns <number> The result of evaluating a curve which represents a spline function.

Param <number> The parameter value at which to evaluate the function.

Crv <curve> The curve representing the function.

OutOfBoundsVal

<number> Value to use if the parameter is out of the parametric range of the curve.

Finally, there are far too many constructors for affine transformations to put them all on a menu. Instead the various classes (e.g., rotations of a vector into a given axis) are put on menus, and extra arguments specify which specific transformation is desired.

translateAxisTransform(*Axis*, *Offset*)

Returns <matDescr> Create a matrix descriptor for translation along an axis.

Axis <keyword> Which axis to translate along.

Offset <number> The value of the translation.

rotateAxisTransform(*Axis*, *Angle*)

Returns <matDescr> Create a matrix descriptor for rotation around an axis.

Axis <keyword> Which axis to rotate around.

Angle <number> Angle of rotation.

scaleAxisTransform(*Axis*, *ScaleValue*)

Returns <matDescr> Create a matrix descriptor for scaling along an axis.

Axis <keyword> Which axis to scale along.

ScaleValue

<number> Scaling value.

rotateAxisToVecTransform(*Axis*, *AxVec*)

Returns <matDescr> Create a matrix descriptor for rotating an axis into a given

vector.

Axis <keyword> Which axis to rotate.

AxVec <geomVector> The vector describing the new axis position.

rotateVectorToAxisTransform(*AxVec*, *Axis*)

Returns <matDescr> Create a matrix descriptor for rotating a vector into an axis.

AxVec <geomVector> The vector to rotate.

Axis <keyword> The axis to which the vector will be rotated.

rotateAxisWithTransform(*Ax1*, *Vec1*, *Ax2*, *Vec2*)

Returns <matDescr> Create a matrix descriptor for rotating an axis into a given vector, with an extra vector to disambiguate the remaining rotational degree of freedom.

Ax1 <keyword> The axis to rotate.

Vec1 <geomVector> The vector to which the axis will go.

Ax2 <keyword> Which axis to use for disambiguating the rotation.

Vec2 <geomVector> The vector to which the second axis will go.

rotateVectorToAxisWithTransform(*Vec1*, *Ax1*, *Vec2*, *Ax2*)

Returns <matDescr> Create a matrix descriptor for rotating an vector into an axis, with an extra axis and vector to disambiguate the remaining rotational degree of freedom.

Vec1 <geomVector> The vector to rotate.

Ax1 <keyword> The axis to which the vector will go.

Vec2 <geomVector> The vector to use for disambiguating the rotation.

Ax2 <keyword> The axis to which the second vector will go.

8.27 GUI Get Functions

Every argument type which any function accessible from the GUI uses must have a "get" function which is used to get arguments which are restricted to the given type. All of these functions accept a string which is supposed to describe the argument (and which usually comes from the menu data structure which is being interpreted by **promptForArgsAndCall**). And they all are wrapper functions for the **getCommon** function which does the work. Only the calling sequences are given for these functions below (except **getCommon**).

getCommon(*ArgDescription*, *ObjectClassString*,*Menu*,*PredicateFns*)

Returns <anything> The result of handling events until the desired kind of argument is received.

ArgDescription
<string> String describing the argument.

ObjectClassString
<string> String name of the object type(s) being requested.

Menu <keyword> Appropriate menu to display when prompting for this kind of argument.

PredicateFns
<listOf keyword> The list of functions to use to test values to see if they are the desired type.

getAngle(**ArgDescription**)
getArc(**ArgDescription**)
getArcOrCircle(**ArgDescription**)
getAxis(**ArgDescription**)
getBoolean(**ArgDescription**)
getCircle(**ArgDescription**)
getCtlMesh(**ArgDescription**)
getCurve(**ArgDescription**)
getEdge(**ArgDescription**)
getGroup(**ArgDescription**)
getInstance(**ArgDescription**)
getLine(**ArgDescription**)
getLineOrPlane(**ArgDescription**)
getNumber(**ArgDescription**)
getInteger(**ArgDescription**)
getObject(**ArgDescription**)
getParametricDir(**ArgDescription**)
getParminfo(**ArgDescription**)
getOrder(**ArgDescription**)
getEndConditionType(**ArgDescription**)
getKnotVector(**ArgDescription**)
getPlane(**ArgDescription**)
getPoint(**ArgDescription**)
getPointOrVec(**ArgDescription**)
getPtArcOrCrv(**ArgDescription**)
getPolyline(**ArgDescription**)
getRbox(**ArgDescription**)
getRboxKeyword(**ArgDescription**)
getRboxFaceKeyword(**ArgDescription**)
getSurface(**ArgDescription**)
getTransform(**ArgDescription**)
getVector(**ArgDescription**)
getFittingParms(**ArgDescription**)
getSplineFn(**ArgDescription**)
getString(**ArgDescription**)
getWindowName(**ArgDescription**)

getObjFromName(ArgDescription)

8.28 C Interface

IOU - C Interface section (esc/rdf)

8.29 Building Shape_edit

IOU - Building Shape_edit section (rdf/tim)

9. Display Manager

This document describes the design and implementation of the Display Manager portion of the **shape_edit** program. It is intended to serve as a guide for implementors of new **shape_edit** display device modules.

9.1 Modeler and Abstract Display

9.1.1 Modeler and Display Relationship

Shape_edit is the model construction portion of Alpha_1, implemented in the Lisp and C languages. A model in **shape_edit** is a set of objects. Model object types include:

- numbers and dimensions,
- basic geometric objects (points, lines, planes, arcs, circles),
- B-spline curves and surfaces,
- solids (primitive spheres, cylinders, etc. and surface shells),
- structures (groups and instances.)

A variety of constructor functions are provided for each type. **shape_edit** provides graphical feedback during the model construction process.

This document describes the Display Manager, which is the portion of **shape_edit** that manipulates the graphical environment on a display device and provides a visual representation of the model geometry during construction and modification.

The model object built by **shape_edit** is represented in the display manager by a set of "display primitives". Display primitives are the conceptual objects to be displayed. They have a type and associated data of a nature specific to that type. Each display primitive can be thought of graphically in terms of an ideal device independent visual representation or description. The display primitives are represented in the physical device memory by one or more "device segments" which are the actual primitives of the device itself. We can think of the display of a model object on a particular device as the creation of device segments on that device which most closely approximate the ideal visual description of the display primitive associated with that object.

9.1.2 Display Primitives

This section discusses the display primitives and their ideal visual representations.

The display primitives include

- point
- polyline
- polygon
- text
- curve
- surface
- mesh

- group
- instance

A point appears on the screen as a small "+" which is one sixty-fourth? of a screen width in size if possible. Points should not appear as single dots at the resolution of the device.

A polyline and polygon are similar in appearance, except that a polyline is not necessarily closed. Both are a series of line segments, drawn at the device's minimum line width. The vertices are not distinguished in any special way, except as the positions where the line changes directions.

Text strings appear in a text size of width 1/80 of the screen width if possible. No particular font is specified.

Curves are displayed as either control polygons or smooth curves or both, depending on flags set from the modeler. In either case, the appearance is like that of a polyline.

Surfaces are displayed as either control meshes or sets of isoparametric curves or both, again based on modeler flags. The appearance of the lines is like that for polylines.

Groups are just collections of other objects, and so have no distinquishing appearance. Instances likewise are just transforms of other objects and so have no unique appearance as instances.

The user manipulates display screens in terms of windows, which are rectangular viewing areas on display devices as shown in Figure 9-1. Although windows are not part of a model, they are manipulated during a modeling session, and may be considered one of the conceptual display primitives in the display manager for the purposes of this discussion. Windows may be overlapped, moved, reshaped or the viewing transformation affecting the geometry visible in the window changed at any time. These window manipulations are local to the display device support code, and are not filtered through **shape_edit**. Higher level operations such as opening windows or copying them between display devices will be handled by **shape_edit**. If a good window package is provided by a workstation, it should be used if possible.

The management of windows varies from device to device and the display device module of the display manager has considerable autonomy in managing windows to encourage local window management interaction. Some things should be consistent on all devices to give consistency to the **shape_edit** user interface:

- Windows can be repositioned or resized at any time using picking/positioning peripherals. Pushing/popping of windows should be supported on devices with opaque windows. Turning windows into icons and back is also desirable. No notification of window changes is given to the Lisp level of **shape_edit**.
- The scale of the view in the window will remain constant on the screen and the "center of attention" will remain in the center of the window as the window is resized. (Scale, translation, and aspect ratio of the viewing transformation will be adjusted to correspond to the new viewport.)
- Viewing transformations may be modified at any time using valuators (knobs, tablet, slider window, etc.) There may be a pick action required to initiate viewing modification or to change which window is affected. No notification to **shape_edit** is required, but provision should be made for querying the viewing matrix from **shape_edit**.
- It is desirable to be able to turn perspective on and off interactively on a per-window basis. When perspective is on, a consistent default view distance or angle will be used throughout Alpha_1.
- Similarly, it is desirable to be able to turn Z clipping on and off when viewing "long" objects.

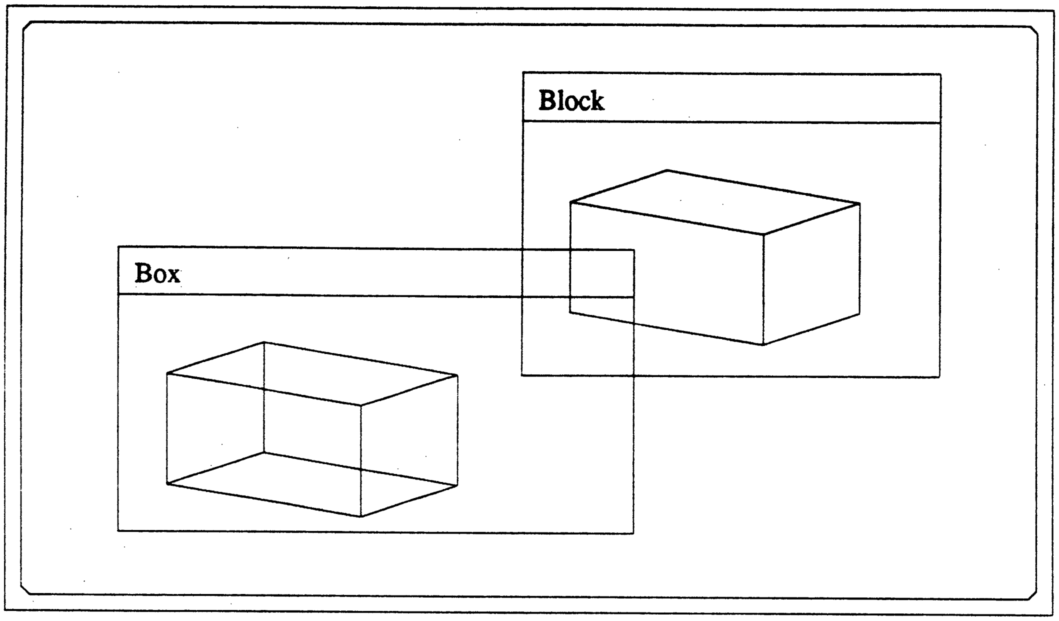


Figure 9-1 Window Appearance

A window object in **shape_edit** contains a window group which is like any other group object. It is simply the set of objects which are visible in the window.

9.1.3 Modeler and Display Data Structure Correspondence

A consistent device model was chosen, tailored to the needs of **shape_edit**. This assumes a display device with a hierarchically segmented, 3D display list. This section describes the correspondence between objects in the modeler and their display device segment counterparts in the abstract display device.

Shape_edit model objects are one-to-one with display list segments stored in a display device. In general, there are corresponding data structures containing the state of the display manager in **shape_edit** and C. Usually, they contain different and complementary information rather than duplicate information. Individual model objects remember the set of display devices on which they are are visible, and handles on their segments on the devices. Group and instance object segments must be able to call other segments. Instance objects must contain a sequence of transformation matrices which are concatenated by the display list follower and applied to geometry.

Display list modification capabilities must include replacing the contents of a segment without

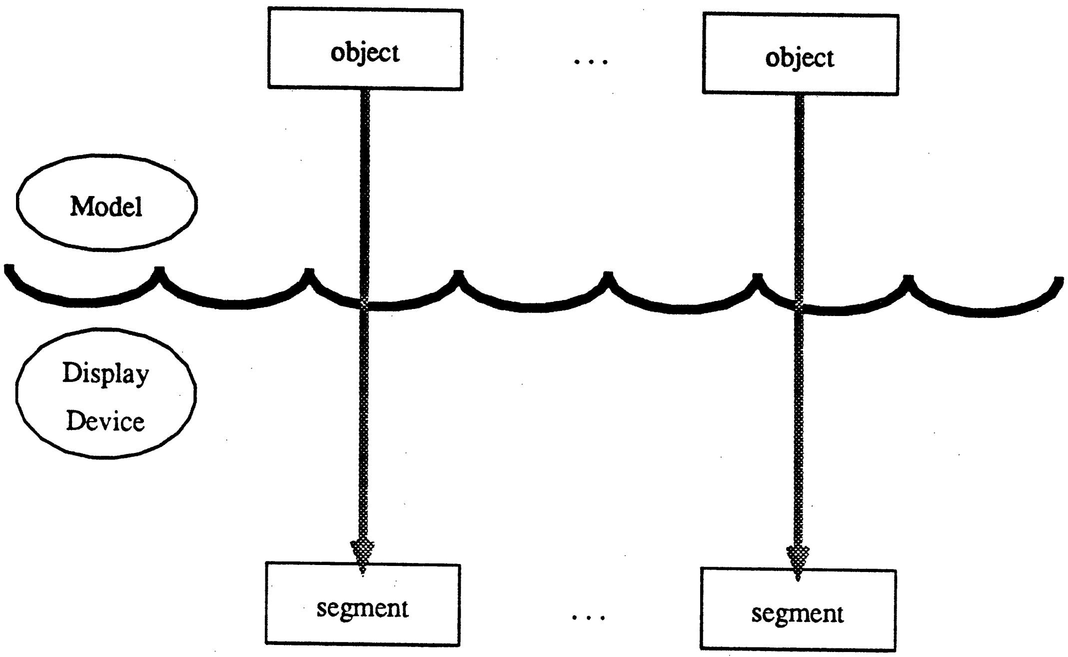


Figure 9-2 Model Objects

changing the segment handle when the geometry of the model object owning the segment is changed. Segment editing operations need only be powerful enough to modify the sequences of matrices and segment calls in group and instance object segments.

A model in **shape_edit** is maintained in parallel on any active display devices. Changes to the model in **shape_edit** cause display segments in the device to be updated.

Figure 9-2 shows how simple objects (e.g., points, polylines, curves, surfaces) are maintained in the display device. Each model object is directly associated with a device segment. If the object is changed, the display manager replaces the entire contents of the segment.

Figure 9-3 shows a group object and its corresponding display device segments. Each object in the group has an associated device segment as described above. The group itself also has an associated segment which simply calls the member segments. This group segment allows objects to be added or deleted from the group by simply adding or deleting calls to the segment objects in the group segment.

Figure 9-4 shows an instance object and its corresponding display device segments. As before, the object being instanced has its own display segment. The instance object transformations are part of the instance segment and may be edited. A matrix can be added to the matrix sequence or deleted from it. A matrix in the matrix sequence can be modified. The object being instanced

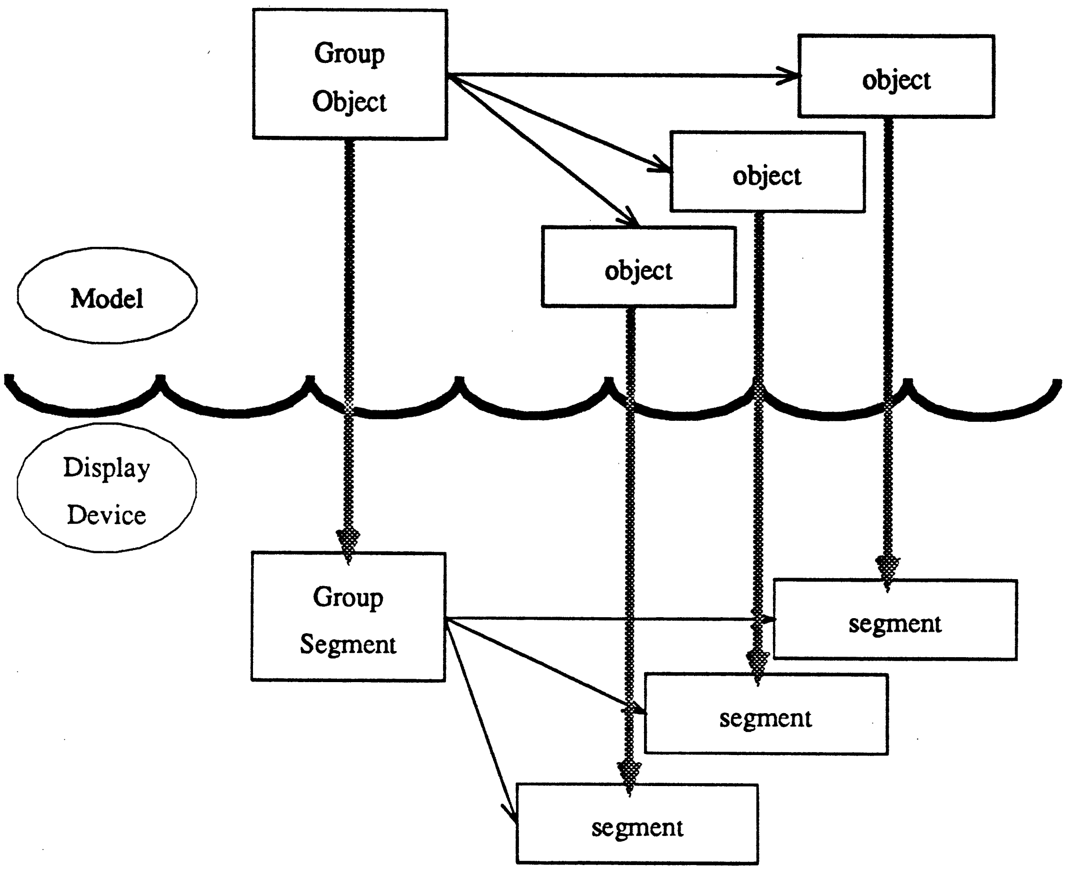


Figure 9-3 Group Objects

may be changed in the display representation by replacing the segment call with a call to another segment.

Figure 9-5 shows a **shape_edit** window object and its corresponding display segments. The window group segment is represented just like any other group, and is called by the window segment. Before calling the window group segment (i.e., before displaying the objects in the window), the window segment must do some setup functions: drawing border and the window name, setting the viewport and the viewing matrix.

Note that Figure 9-5 shows the main display list of the device. Only the window segments are called directly by the display list. All the other segments are drawn indirectly by segment calls in the window group segment.

Windows are created in the device by creating a window segment with an empty window group

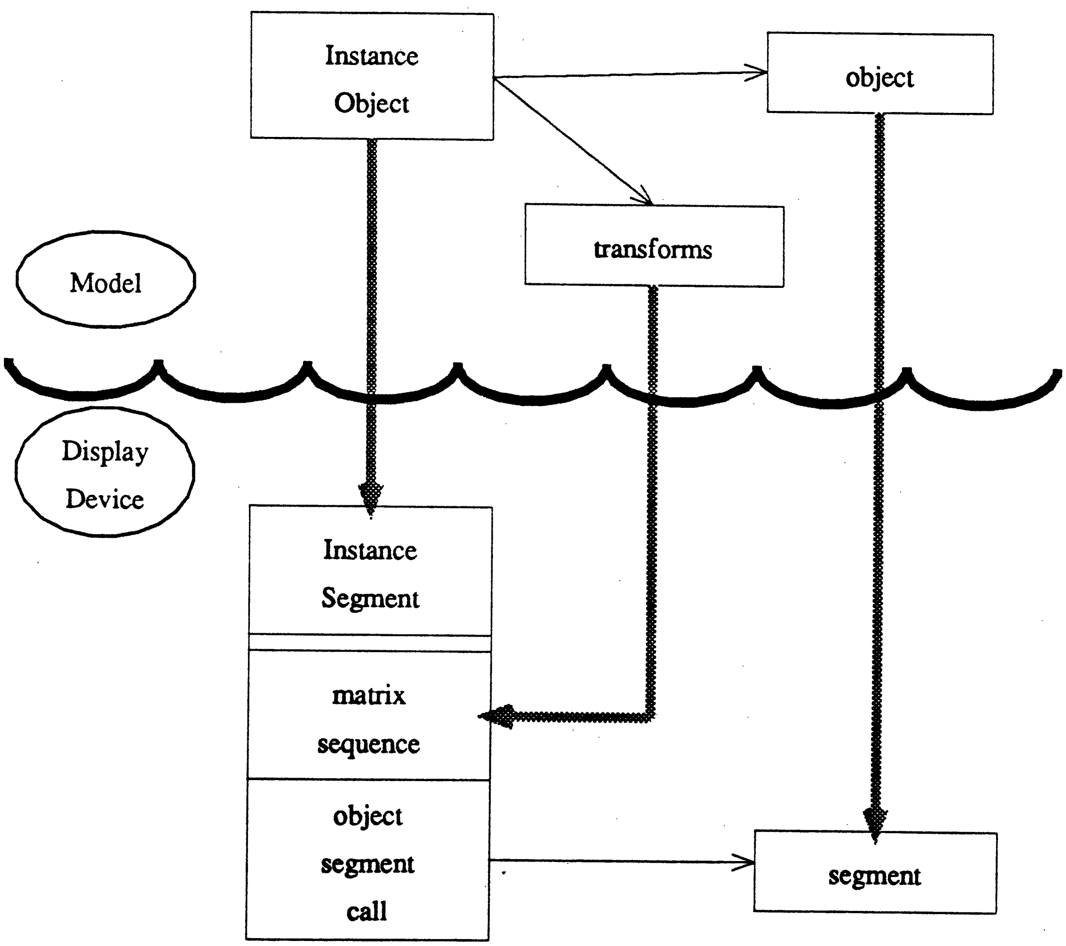


Figure 9-4 Instance Objects

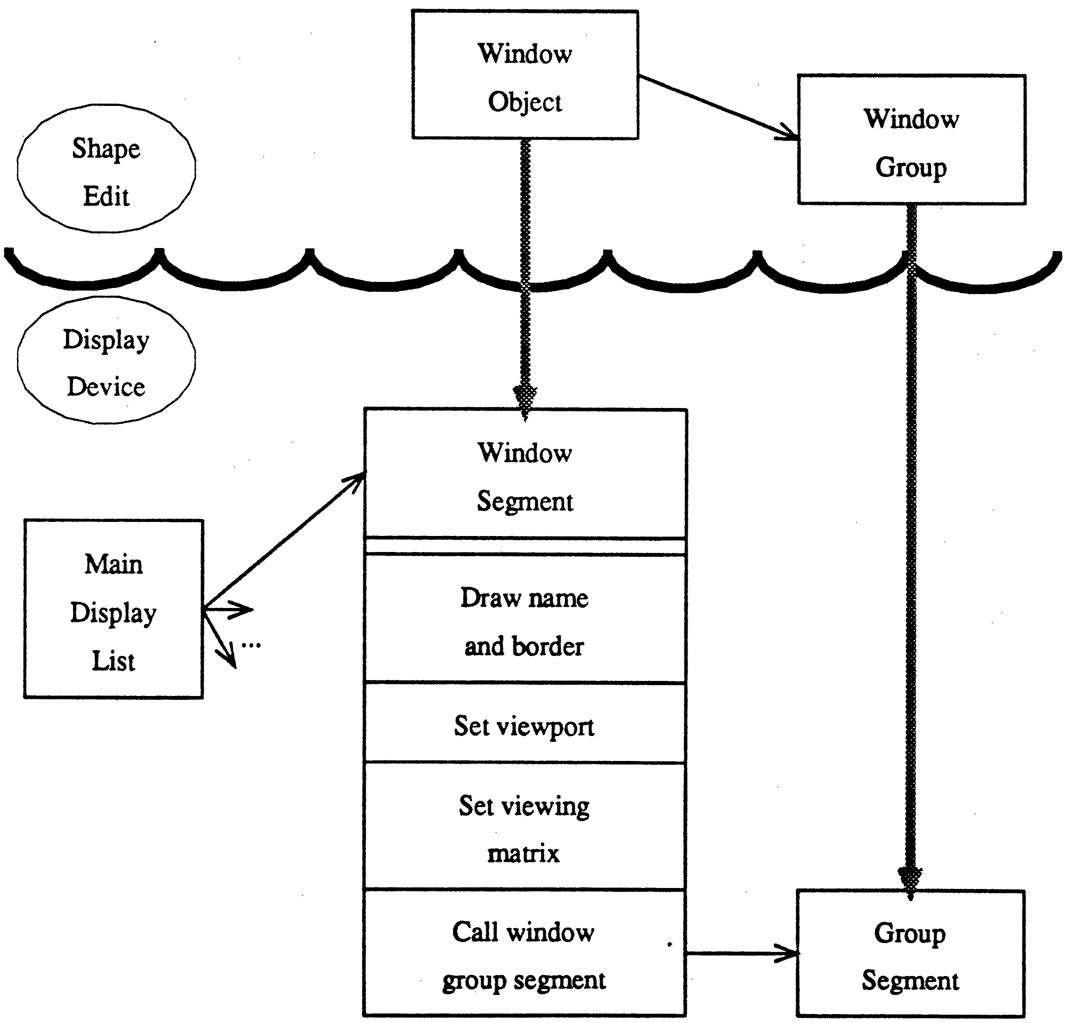


Figure 9-5 Window Objects

segment (no segment calls). A window can be removed by removing it from the main display list. Objects within the window are shown (or unshown) by adding (or removing) an appropriate segment call in the window group segment.

9.2 Implementation Design

9.2.1 Levels of Implementation

The design of the Display Manager is described here to provide context for understanding what is required when adding a new display device module. Design principles included:

- Portability: As little new code as possible should be needed to add a new display device. Interface mechanics should be handled automatically to speed implementation of new devices and avoid errors in maintenance of old ones.
- Modularity: There should be a clear and clean, standard boundary around the support for each display device, without adding special cases into the base system.
- Consistency: The effects visible to the user as well as the structure of the code should be as similar as possible from device to device.
- Efficiency: High performance display devices should be supported to their full capability, without being penalized by the other considerations.
- Generality: A wide range of good display devices should be supported well.
- Simplicity: As little state as possible is kept in the display manager, avoiding duplicate maintenance. State is kept as close as possible to where it will be used, and in a form where it is directly usable to minimize transfer overhead. A minimal set of operations which can get the job done is used.
- Flexibility: Not only should **shape_edit** support multiple display devices, it should be able to support two or more simultaneously during a single design session. This allows using simple workstation displays for low-bandwidth work and sharing high performance displays as desired.

All display devices must meet the functional requirements described in the section on display primitives (see section 9.1.2 [Display Primitives], page 203) with some combination of software and hardware capabilities. Figures 9-2 to 9-5 showed two levels of the display manager: the **shape_edit** modeling environment, and the display device. In the display manager implementation, there are actually several more layers between the **shape_edit** and the display device as shown in Figure 9-6.

These extra layers are necessary to achieve the design principles stated at the beginning of this section. We usually lump the display device library layer together with the device hardware and think of three conceptual levels: the **shape_edit** modeler, the C display manager interface, and the display device. The C display manager level contains both device independent code ("nearest" the **shape_edit**) and device dependent code ("nearest" the display). The entire C layer serves as an adjustable insulation between the wide variety of display devices which are available and the uniform interface which **shape_edit** must use to communicate with them. For a device which closely matches our notion of abstract display device capabilities, the C layer will be minimal. For a device such as a bitmap display that does not provide these capabilities, the C layer will actually simulate a display list device, and will contain much more code.

The layer of device independent C functions provides dispatching to the display device dependent C functions. The C provides a generic display list segment creation operation, so in essence only a single C function needs to be called from **shape_edit** to display all types of object.

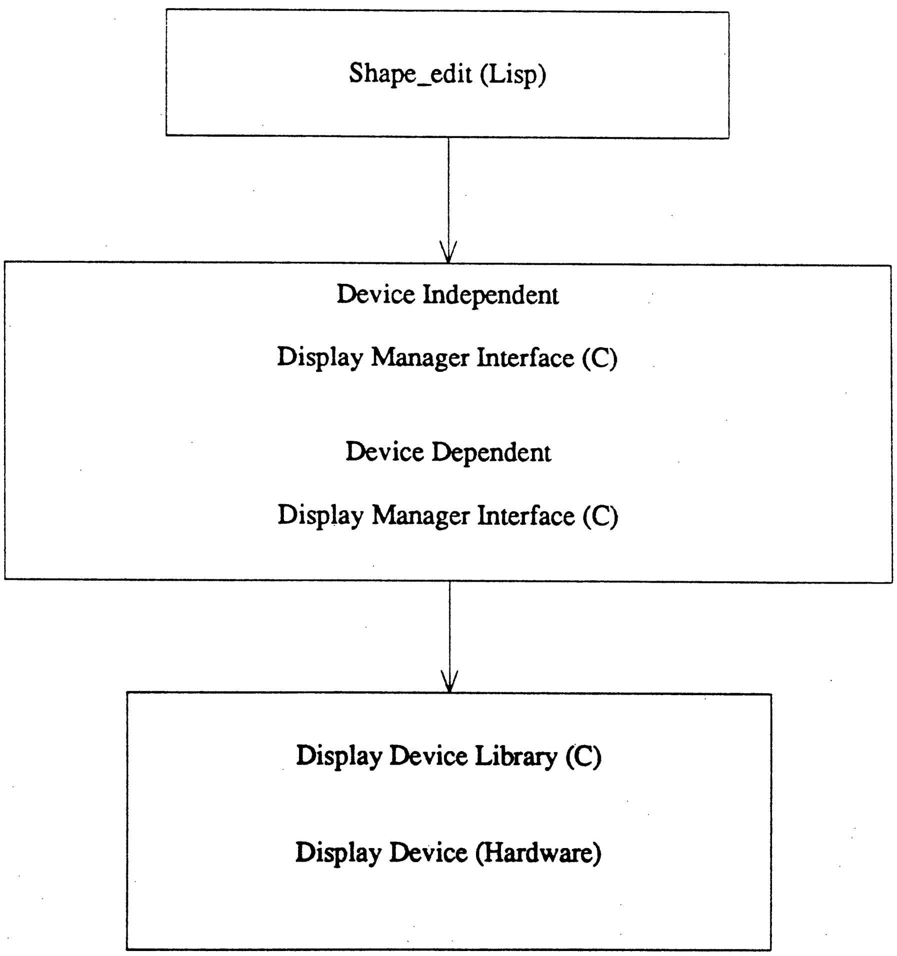


Figure 9-6 Display Levels

LINKS	
ATTRIBUTES	
string	**text**
point	**position**

Figure 9-7 Text_string Data Structure

The device dependent layer consists of the operations which are invoked by the **shape_edit** display manager code in PSL, through the device dispatching level. A file of function bodies for each device is merged into dispatcher calling sequences.

The local operations, e.g., for window management on the device, may be programmed in a processor in the display device, in a separate Unix process, or in an interrupt handler linked into **shape_edit**.

9.2.2 Data Structures for Displayable Objects

This section gives an overview of the C data structures with which you will need to be familiar to implement display manager support for a new device. Some of the objects described are "ephemeral" in the C code, meaning that the structures are created merely for handing data from PSL into C. These structures will be freed as soon as control returns to the PSL code. The other objects are "persistent" C data structures which remain in the C malloc heap throughout the **shape_edit** session once they are created.

Text_string objects contain a text string and the position at which it is to be displayed (Figure 9-7). It should not be necessary for you to manipulate this data structure, except to access the fields.

A point_obj contains a single point which may be in any of a number of coordinate spaces. It has a tag which indicates its type (Figure 9-8). You will probably want to coerce all point_obj structures to regular Euclidean 3-space points. The "point" struct contains fields for X, Y, and Z coordinates, and the coercion can be done using:

LINKS	
ATTRIBUTES	
pt_type	**pt_tag**
mat_element	**coords[]**

Figure 9-8 Point Data Structure

```
point_obj * pt;     /* Point object to be converted. */
point send_point;   /* Return value declaration. */

coerce_to_e3( &send_point, pt->coords, pt->pt_tag );
```

The coordinates of the result can be accessed as

```
send_point.X
send_point.Y
send_point.Z
```

A polygon object consists of a number of contours, each of which has a number of vertices (Figure 9-9). The contours are stored as a doubly-linked list, and the vertices of each contour are a circular doubly-linked list. Each vertex contains an embedded point struct, so coercion to E3 is not necessary. An example of code for tracing through the polygon object is given in the description of the polygon display routine in the section on Operations on Display List Segments below (see section 9.2.4 [Operations on Display List Segments], page 219).

The disp_pts structure is an ephemeral struct for passing groups of points packed into an array into the C code (since the C polygon and polyline structures contain linked lists). The structure, shown in Figure 9-10, contains a tag like the point_obj tag to indicate the dimension of the points. All the points will be the same size, and will need to be converted to E3 using the coerce_to_e3 routine. An example of code for accessing the points is included with the description of the disp_pts display routine in the section on Operations on Display List Segments below (see section 9.2.4 [Operations on Display List Segments], page 219).

Crv_object data structures are merely wrappers for an embedded curve data structure which con-

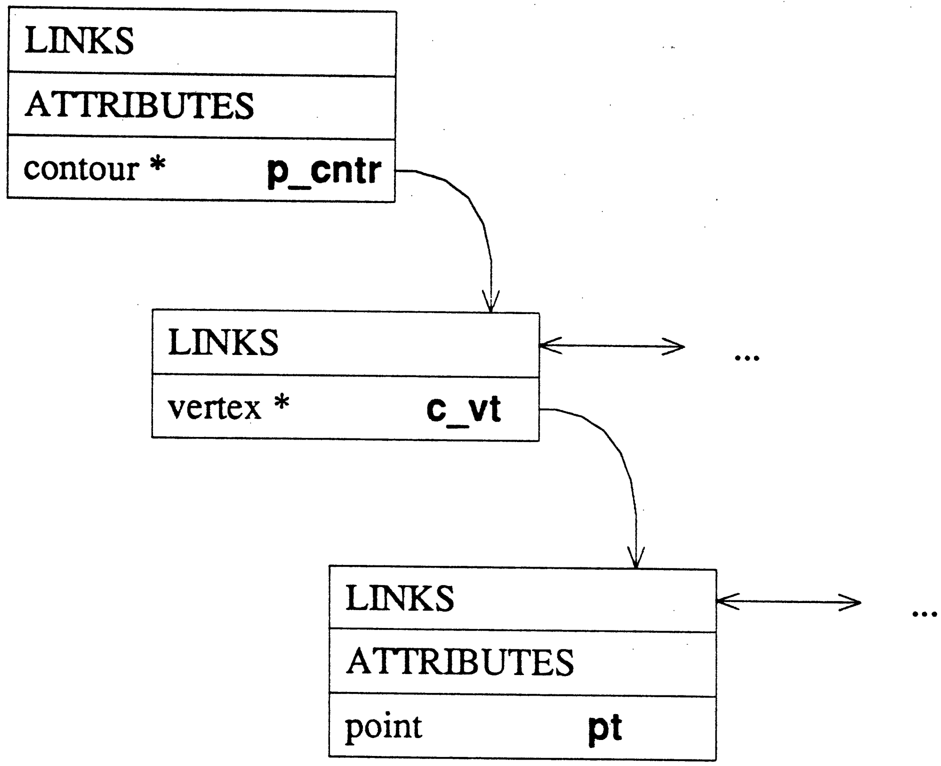


Figure 9-9 Polygon Data Structure

tains the spline data (Figure 9-11). Srf_object data structures are a similar wrapping for an embedded surface structure (Figure 9-12). If you implement the display manager curve and surface display routine in the simplest way described later on in this document, you should not need to access any of the fields of either object. If your target device contains special curve or surface display operations which you wish to take advantage of, you will have to get more information about manipulating these C structures. It is highly recommended that you use the simple method first in any case.

Disp_group objects are ephemeral structures which convey a block of dl_seg references to the display code (Figure 9-13). The dl_seg references are to objects which are members of a **shape_edit** group object, and which have already been sent to the display device. The segs field is a pointer to an array of dl_seg pointers.

The disp_instance, instance_mat, and disp_matdescr objects are used in displaying **shape_edit**

int	**t_tag**
pt_vector *	**coords**

Figure 9-10 Disp_pts Data Structure

LINKS		
ATTRIBUTES		
curve **c**		
	int	**c_order**
	ec_type	**c_type**
	knot_vector *	**c_kv**
	ctl_polygon *	**c_poly**

Figure 9-11 Curve Object Data Structure

LINKS	
ATTRIBUTES	
surface	**s**
int [2]	**s_orders**
ec_type [2]	**s_types**
knot_vector * [2]	**s_kvs**
ctl_mesh *	**s_mesh**

Figure 9-12 Surface Object Data Structure

int	**t_tag**
int	**n_segs**
seg_block *	**segs**

Figure 9-13 Dgroup Data Structure

instance objects (Figure 9-14). Disp_instance objects just serve as a base for a list of disp_matdescr objects. They are persistent objects, remembered by the C_Address slots of **shape_edit** instance objects.

Disp_matdescr objects are similarly persistent C representations of a class of **shape_edit** object types such as rotateX, translateXYZ, and so on. All object types of that class are translated to a single C type, disp_matdescr objects, which have a mat_type slot that differentiates them and a variable number of floating-point parameters.

Unique among the C data structures, the instance_mat and disp_matdescr objects are not device dependent, and thus a single copy is shared among all devices on which an instance object is visible.

9.2.3 Display Manager Data Structures

Dl_seg

Each display primitive that has been displayed on a device is represented in the display manager by a persistant C structure known as a **dl_seg**. C dl_seg objects are persistent C data structures, acting as "handles" on the chunk of display list memory in a particular display list device. On devices where the display list is simulated by C code, such as bitmap devices, dl_seg objects are the elements from which display lists are constructed. A diagram of the dl_seg is shown in Figure 9-15.

The most important slot of dl_seg objects for display list devices is the seg_num, a long integer of up to 32 bits which is used to identify the particular segment of display list memory to the device. The display device support must handle generation of the segment numbers, insuring that they are unique within the device module.

Several slots are provided to support the user interface implementation: The seg_type slot is intended to support pick sensitivity classified by an object type identifier, probably the t_tag values of the geometry objects. The highlit slot is a boolean which indicates whether a particular object is highlighted on the display to show it is selected. The pick_id slot is an optional slot which may be used by the display device module to contain integer identifiers used in picking in addition to the seg_num if necessary.

There is an optional disp_rep slot which could contain a pointer to any Alpha_1 C object type when it is necessary for the device module to cache a copy of the geometry for future reference. It is intended that the bitmap display devices use this to build their simulated display lists, and that real display list devices will probably not need to maintain this cache.

Finally, an optional seg_size field may be useful for remembering generated segment sizes for objects on devices which can overwrite segments only with things the same size or smaller.

Window_object

A diagram of the C window_object is shown in Figure 9-16.

C window_object objects are persistent data structures, holding the state of a window on a display device in the C malloc heap. They contain a string which is the name of the window, and a pointer to a dl_seg object for the window_group which contains the set of objects visible in the window. A set of optional slots are also available for use by the window management logic for the device, but need not be used if not needed.

Display_dev

C display_dev objects, shown in Figure 9-17 contain a string name, as well as two boolean flags. The open flag indicates whether the device is active, and the initialized flag differentiates between

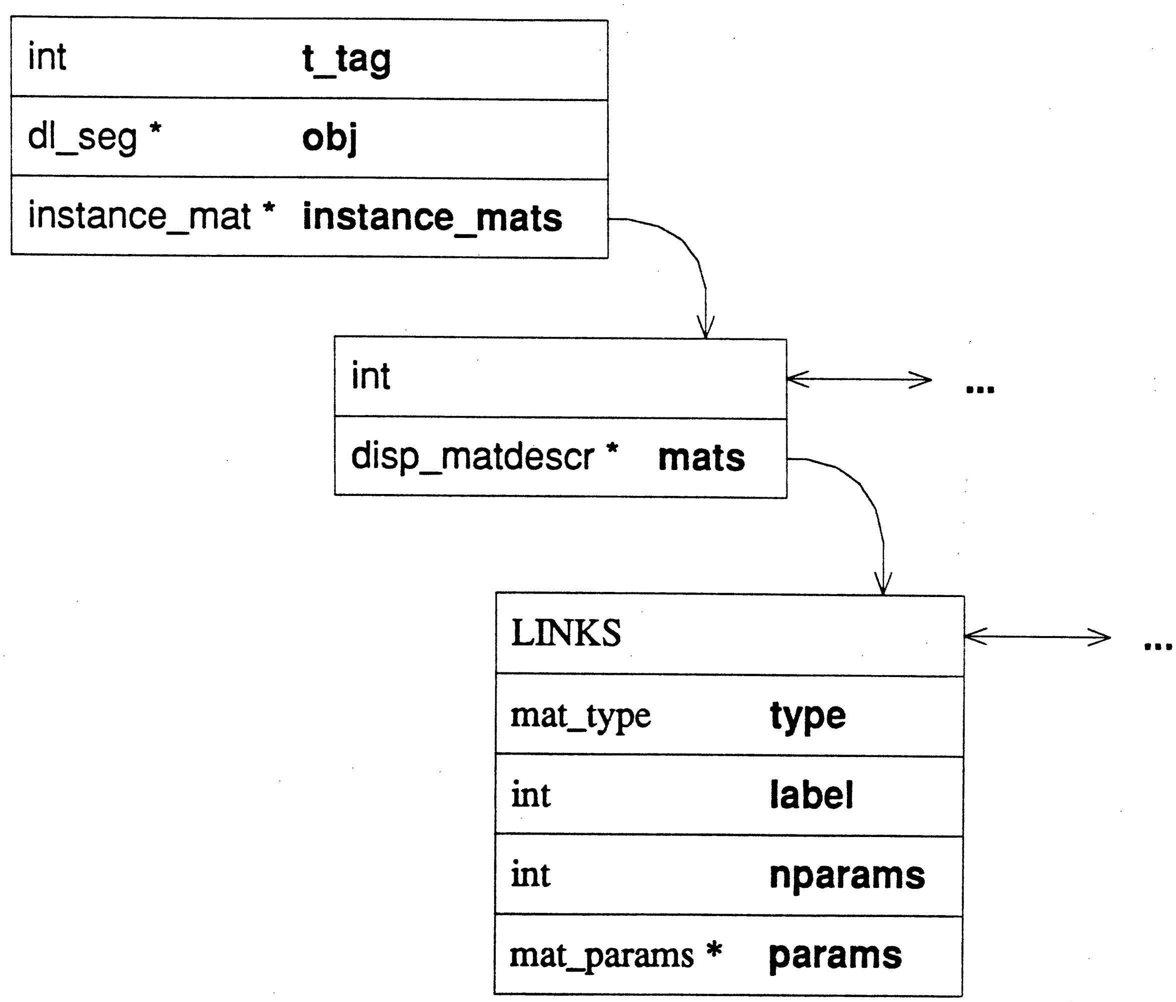


Figure 9-14 Dinstance Data Structure

LINKS	
ATTRIBUTES	
long	**seg_num**
rgb *	**seg_color**
int	**seg_type**
int	**pick_id**
object *	**disp_rep**
int	**seg_size**

Figure 9-15 Dl_seg Data Structure

the first time a device is grabbed during a **shape_edit** session and the times it is regrabbed. A window_list of window_objects should be accumulated for use in dropping and regrabbing the device.

The dev_fns slot of the display_dev structure supports dispatching of generic display operations to specific functions for the "current display device".

Other fields are optional and may be used to control the window management on the device if needed.

9.2.4 Operations on Display List Segments

The two most basic display object operations are; 1) creating a device segment for a display object and 2) replacing the data in an existing device segment.

The replacement of data for an existing device segment is accomplished by overwriting the existing segment with totally new data for the same type of display object. Since overwriting a device segment with new data for a display object is, in practice, a nearly identical operation to the initial creation of the device segment for the display object, both these operations are combined into a

LINKS	
ATTRIBUTES	
string	**name**
dl_seg *	**window_group**
dl_seg *	**highlight_group**
int	**min_x, max_x, min_y, max_y**
float[4][4]	**window_xform**
float[4][4]	**view_xform**
float[4][4]	**save_view**
dl_seg *	**window_frame**
long	**window_num**

Figure 9-16 Window_object Data Structure

single routine for each display object type. The type of device segment to be created or overwritten is implicit in each of these routines.

The calling sequence for all the create/replace segment operations are similar. Below is an illustration of the general form. (For the remainder of this document, the string "devnm" will refer to the actual device abbreviation. In this chapter, device specific routine names will contain "devnm" in place of the actual device name.)

*devnm*_dls_*xxx*(*obj_arg*, *seg_arg*)

The *obj_arg* provides the data necessary to create or overwrite a device segment for the given object type. The *seg_arg* defines the type of operation required. If the *seg_arg* is NULL, then the display operation is to create a new device segment of the implied type. The data to be used in the creation of the new device segment is provided by the first argument. In addition, a dl_seg

int	**t_tag**
string	**name**
boolean	**open, initialized**
window_object *	**window_list**
dev_dispatch *	**dev_fns**
float	**coord_aspect**
int	**min_x, max_x, min_y, max_y**
window_object *	**curr_window**
menu_object *	**menu_list**
dl_seg *	**curr_segment**

Figure 9-17 Display_dev Data Structure

C structure must be created as a handle on the new device segment. It is the responsibility of each device module to fill the seg_num slots of any dl_seg structs which it creates. The seg_num is unique within the context of that particular device module, and must in some way identify the segment within the device.

If the second argument is not NULL, then the argument is a pointer to a dl_seg structure that provides a handle on an already existing device segment of the implied type. In this case, the existing device segment is to be overwritten with the new information provided by the first argument.

In either case, a pointer to the dl_seg C structure is returned by the routine. In this way the calling function doesn't need to know whether a new dl_seg was constructed or not. In the case of a create segment operation, a pointer to the created dl_seg structure is returned. In the case of a replace segment operation, the return is the same as the seg_arg.

Since segments must be created for each type of object, it is usually simpler to encapsulate the segment setup and updating code in two routines which are called at the beginning and end of each devnm_dls_xxx routine. The setup routine generally determines the segment number to be used, while the update routine prepares the dl_seg structure which must be returned. An outline for the setup routine might be:

```
long
devnm_start_draw( seg )
dl_seg * seg;

    int segment_number;

    if ( seg )        /* Segment already exists, use same number. */
        segment_number = seg->seg_num;
    else
        segment_number = /* Generate new segment number to use. */

    /* Perform whatever operations are needed to open the
     * device segment.
     */
    return segment_number;
```

An outline of the update routine might be:

```
dl_seg *
devnm_end_draw( seg, segment_number )
dl_seg * seg;
long segment_number;

    /* Perform whatever operations are needed to close the
     * device segment.
     */

    /* Create dl_seg if object is displaying for the first time. */
    if ( seg == NULL )

        seg = new_dls();
        seg->seg_num = segment_number;


    /* Return new or updated segment structure. */
    return seg;
```

The names of these routines and their calling sequences will be used as examples for the code outlines in the remainder of this section, but the actual names and calling sequences for your device may be different.

The remainder of this section describes the particular display routines for each display primitive.

Point

The **devnm_dls_pt** declaration is:

Move Draw Draw ...

Figure 9-18 Polyline Segment

```
dl_seg *
devnm_dls_pt( pt, seg )
point_obj *pt;
dl_seg *seg;
```

An outline of the function body code might be:

```
segment_number = devnm_start_draw( seg );

/* Send commands for display of point. */

return devnm_end_draw( seg );
```

Polyline

The device segment for a polyline should consist of a single "move,draw,draw,..." sequence as shown in Figure 9-18. The move is to the coordinates of the first point in the pt_vector of the polyline display object. The subsequent draws are to the coordinates of each of the remaining points in the pt_vector.

The **devnm_dls_dpts** declaration is:

```
dl_seg *
devnm_dls_dpts( pts, seg )
disp_pts * pts;
dl_seg * seg;
```

An outline of the code might be:

```
pt_vector * pts = pts_obj->coords;    /* Declarations. */
mat_element * v_ptr;
point send_point;

segment_number = devnm_start_draw( seg );

num_verts = pt_vec_size( pts );    /* Figure out how many points. */

v_ptr = pts->value;
for ( i=0, i < num_verts; i++, VEC_NEXT( v_ptr, pts ) )

        /* Change point to E3 */
```

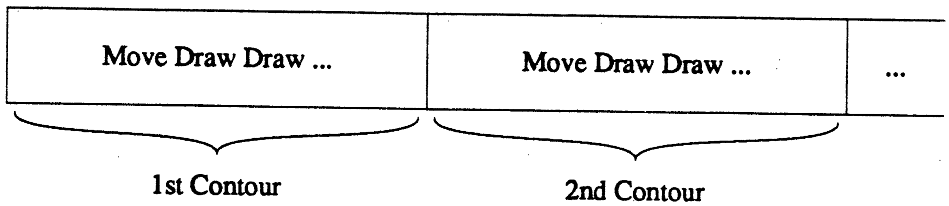


Figure 9-19 Polygon Segment

```
        coerce_to_e3( &send_point, v_ptr, pts->pt_tag );

        /* Send send_point.X, send_point.Y, send_point.Z */

    return devnm_end_draw( seg, segment_number );
```

Polygon

Polygons are similar to polylines (above) except that the last draw goes back to the first point to close the figure. Polygons may also contain multiple contours, so the segment may appear as diagramed in Figure 9-19.

The calling sequence for **devnm_dls_poly** is:

```
dl_seg *
devnm_dls_poly( pg_ob, seg )
polygon * pg_ob;
dl_seg * seg;
```

An outline of the code might be:

```
contour * c_ptr;
vertex * v_ptr;

segment_number = devnm_start_draw( seg );

/* Send each contour */
TRACE( c_ptr, poly->p_cntr )

    /* Count points in contour; start with 1 to duplicate first point. */
    num_verts = 1;
    TRACE_CIRCULAR( v_ptr, c_ptr->c_vt )
            num_verts++;
```

Move Draw Draw ...

Figure 9-20 Curve Segment

```
/* Send points. */
TRACE( v_ptr, c_ptr->c_vt )

     /* Coordinate access is: v_ptr->pt.X
      *                       v_ptr->pt.Y
      *                       v_ptr->pt.Z
      */
```

```
return devnm_end_draw( seg, segment_number );
```

Text

The **devnm_dls_text_str** calling sequence is:

```
dl_seg *
devnm_dls_text_str( text, seg )
text_string * text;
dl_seg * seg;
```

Curve

The **devnm_dls_crv** declaration is:

```
dl_seg *
devnm_dls_crv( crv, seg )
crv_object * crv;
dl_seg * seg;
```

The **generic_draw_curve** routine is provided for devices that do not have built in curve drawing capability or where such capability is unsuitable. If the **generic_draw_curve** routine is to be used, then the **devnm_dls_draw_mesh** (below) routine must be implemented. The resulting device segment will resemble the polyline device segment as shown in Figure 9-20, with the point sequence representing the refined curve control polygon.

An outline of the code for **devnm_dls_crv** is then

```
segment_number = devnm_start_draw( seg );
```

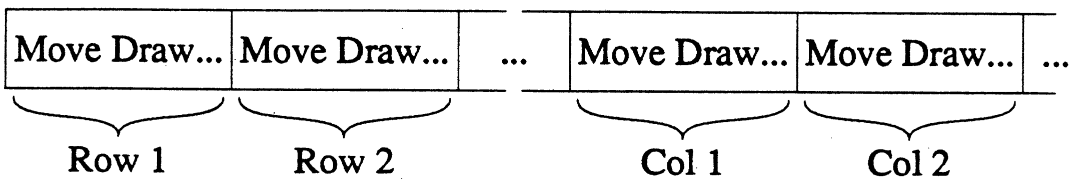


Figure 9-21 Surface Segment

```
generic_draw_curve( crv );

return devnm_end_draw( seg, segment_number );
```

Even if the device's curve drawing capability is to be used eventually, it is highly recommended that **devnm_dls_crv** first be implemented using **generic_draw_curve** and **devnm_dls_draw_mesh**.

Surface

The **devnm_dls_spl** declaration is:

```
dl_seg *
devnm_dls_spl( srf, seg )
srf_object * srf;
dl_seg * seg;
```

The **generic_draw_surface** routine is provided for doing surface refinement on the host, and then displaying the refined surface mesh on the device. If the **generic_draw_surface** routine is to be used, then the **devnm_dls_draw_mesh** (below) routine must be implemented. The resulting device segment will have a sequence resembling the polyline device segment for each row and column of the refined surface control mesh, as shown in Figure 9-21.

An outline of the code for **devnm_dls_spl** is then

```
segment_number = devnm_start_draw( seg );

generic_draw_surface( srf );

return devnm_end_draw( seg, segment_number );
```

Even if surface refinement is to be done eventually on the device, it is highly recommended that **devnm_dls_spl** first be implemented using **generic_draw_surface** and devnm_draw_mesh.

Mesh

The **devnm_draw_mesh** declaration is:

Add data to a device segment for visualizing a single row or column of a mesh display object.

This routine need only be implemented if the **generic_draw_curve** or the **generic_draw_surface** routines are to be used.

Declaration:

```
void
devnm_draw_mesh( mesh, dir, indx )
pt_vector * mesh;
array_dir dir;
int indx;
```

The dir argument is one of either ROW or COL which are defined constants. The index argument gives the particular row or column of interest.

This routine is unlike other display object operations, in that it is always called from within the context set by the **devnm_dls_spl** (or **devnm_dls_crv**) routine. That routine calls **generic_draw_surface** (or **generic_draw_curve**) which generates the refined control mesh and dispatches to the routine **devnm_dls_draw_mesh** for depositing of the appropriate data into device segment.

Each call to the **devnm_dls_draw_mesh** routine adds a sequence resembling the contents of the polyline device segment to the "open" device segment.

An outline of the code might be as follows:

```
int size, i;
mat_element *mat_el;
point send_point;

/* Get the size of a row or column. */
size = mat_size( mesh, otherdir(dir) );

/* Send over the row or column of points one at a time. */
for( i = 0, mat_el = mat_index( mesh, 2, DIR_SUBSC( dir, indx, 0 ) );
     i < size;
     i++, MAT_NEXT( otherdir(dir), mat_el, mesh )  )

    /* Change point to E3. */
    coerce_to_e3( &send_point, mat_el, mesh->pt_tag );

    /* Send send_point.X, send_point.Y, send_point.Z */
```

Note that the usual segment setup and closing routines are not called by **devnm_dls_draw_mesh**.

It is possible that the paradigm described above may not readily fit the architecture of a device. For example, it may be that a particular device requires that individual "move,draw,draw..." sequences be placed in individual segments. In this case the segment for the surface must be structured as an outer segment, that calls all "sub-segments" individually created by the **devnm_dls_draw_mesh** routine. The **devnm_dls_draw_mesh** routine will then need to communicate to the **devnm_dls_spl** routine the identifiers for these sub-segments. A C structure known to both the **devnm_dls_spl** and the **devnm_draw_mesh** routines could be used for storing device segment identifiers. In such a case the disp_rep structure for the dl_seg associated with the surface display object will need to be used in order to keep handles on the sub-segments created by the devnm_dls_draw_mesh routine.

Group

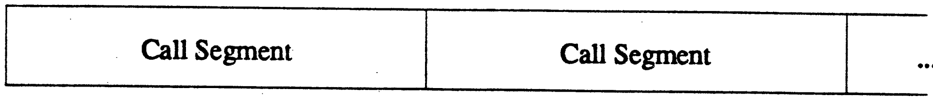


Figure 9-22 Group Segment

The **devnm_dls_dgroup** declaration is:

```
dl_seg *
devnm_dls_dgroup( group, seg )
disp_group * group;
dl_seg * seg;
```

The disp_group provides a block of dl_seg structure pointers. A segment must be created that calls the other segments to which the dl_seg structures refer. The resulting device segment will consist of a series of calls to other device segments as shown in Figure 9-22.

Insertion of a segment call into the group must be accomplished in such a way so as to make possible later editing of the group segment. Specifically it must be possible to delete a call to a specific segment within the group segment by specifying the seg_num of segment called. In general this will be accomplished by labeling the segment call within the group segment with a label based on the seg_num of the segment called. Many devices provide such a capability through the use of segment labels. Segment labels provide handles to locations **within** a given segment for use by the segment editing commands of the device.

An outline of the code for **devnm_dls_dgroup** might be:

```
segment_number = devnm_start_draw( seg );

/* Call each member segment. */
for( i = 0; i < group->nsegs; i++ )


    /* Get the segment number for the device segment to include.*/
    segment_num = (*group->segs)[i]->seg_num;

    /* Call the segment indicated. */


return devnm_end_draw( seg, segment_number );
```

The **devnm_dls_add_object_to_group** declaration is:

```
void
devnm_dls_add_object_to_group( group, obj )
dl_seg * group;
dl_seg * obj;
```

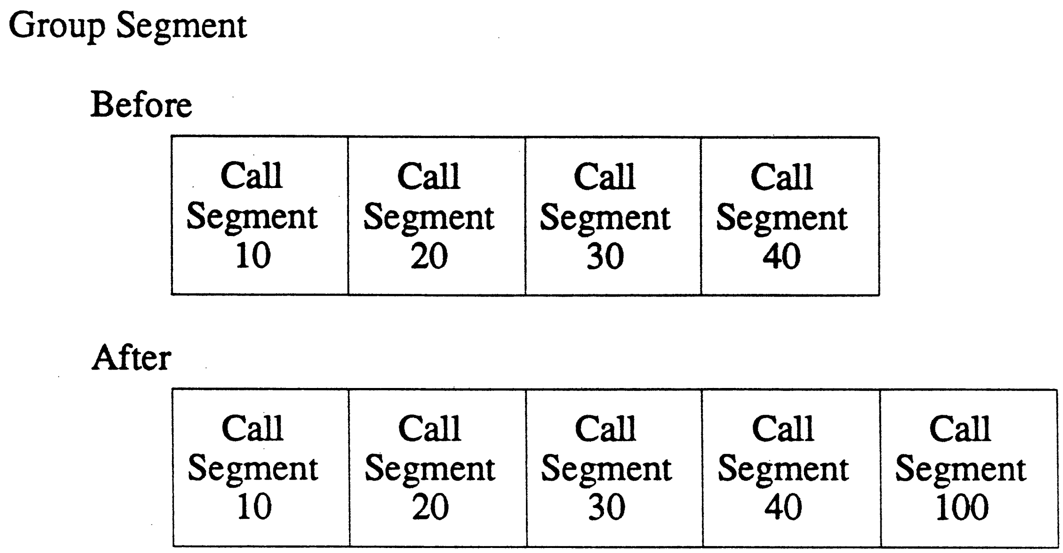


Figure 9-23 Dls_add_to_group

This routine adds a segment call to an existing group segment. The group argument gives the dl_seg of the already existing group segment to be modified. The object argument gives the dl_seg handle on the already existing segment which is to be called by the group segment.

The ordering of calls within a group segment is irrelevant and the new segment call may be added any place within the group segment that is convenient, typically at the end of the segment, as shown in Figure 9-23.

The devnm_dls_remove_object_from_group declaration is:

```
void
devnm_dls_remove_object_from_group( group, obj )
dl_seg * group;
dl_seg * obj;
```

This routine removes a specified segment call from an existing group segment, as shown in Figure 9-24. The arguments are as above for **devnm_dls_add_object_to_group**.

The seg_num field of the dl_seg structure pointed to by the obj argument is used as a basis for locating the segment call to delete from the group segment. This is typically done by a labeling scheme, as described above for devnm_dls_group.

Instance

The *devnm_dls_dinstance* declaration is:

```
dl_seg *
devnm_dls_dinstance( instance, seg )
disp_instance * instance;
```

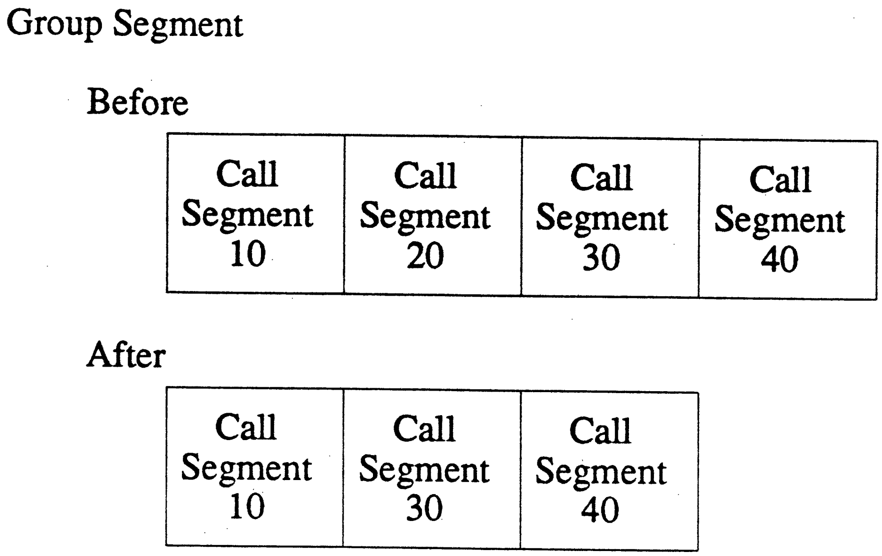


Figure 9-24 Dls_remove_from_group

dl_seg *seg;

The instance argument is a pointer to a disp_instance structure which gives a handle on the dl_seg structure for an existing device segment that will be called within the instance segment. This structure also gives a pointer to the head of the list of disp_matdescr structures containing the necessary information to form the matrices in the instance segment. See section 9.1.2 [Display Primitives], page 203 for a description of the disp_instance structure and examples of accessing the disp_matdescr list.

Under certain circumstances, it may be convenient or necessary to have the disp_instance structure pointed to by the disp_rep of the instance dl_seg.

An instance segment conceptually contains a list of transformation matrices, followed by a call to the segment for the display object that is being instanced, as shown in Figure 9-25. The device matrices specify modeling transformations that are to be applied to data in the called segment.

The effect of the matrices within the segment shown in Figure 9-25 should be understood as proceeding from more global to more local as we go from left to right. That is matrix0 is first applied to the data in the called segment, then matrix1 and so forth, that is:

```
transformed data = data * matrix0 * matrix1 * ... * matrixN-1
```

This ordering corresponds to the matrix stack operations usually found in most display devices. That is, as the device segment is traversed by the display list follower, the matrix elements are typically pre-concatenated.

The instance segment must be constructed so as to allow arbitrary editing of the list of matrices it contains. Deletion of specified matrices from the matrix list as well as insertion of new matrices at a specified location in the matrix list must be possible.

Figure 9-25 Instance Segment

If your device has a segment labeling scheme, then the value provided by the modeler system in the label field of each disp_matdescr C structure may be used as a basis for a label to identify the segment matrix element location within the constructed instance segment.

The instance segment must also be constructed so as to allow the segment call within the instance segment to be replaced by a call to a different segment.

The *devnm_dls_replace_instanced_obj* declaration is:

```
void
devnm_dls_replace_instanced_obj( instance, obj )
dl_seg * instance;
dl_seg * obj;
```

The segment call within the instance segment is replaced by a call to a different segment, as shown in Figure 9-26.

The *devnm_dls_insert_instance_matdescr* declaration is:

```
void
devnm_dls_insert_instance_matdescr( instance, where, matdescrs )
dl_seg * instance;
disp_matdescr * where;
disp_matdescr * matdescrs;
```

This routine inserts a list of matrices into the matrix list of an existing instance segment. The matdescrs argument is a pointer to the first disp_matdescr in a list of disp_matdescrs that provide the data needed to specify the new matrices to be inserted.

The where argument is used to specify the location in the matrix list where the new list is to be inserted. This argument is a pointer into the the list of disp_matdescr structures that is maintained by the modeling system for this instance object. The disp_matdescr structure pointed to corresponds to the matrix to insert **before** in the matrix list in the device segment. If the where argument is NULL then insertion at the end of the list of matrices in the segment is specified. A diagram of the operation is shown in Figure 9-27.

The *devnm_dls_delete_instance_matdescr* declaration is:

```
void
devnm_dls_delete_instance_matdescr( instance, first, last )
dl_seg * instance;
disp_matdescr *first, *last;
```

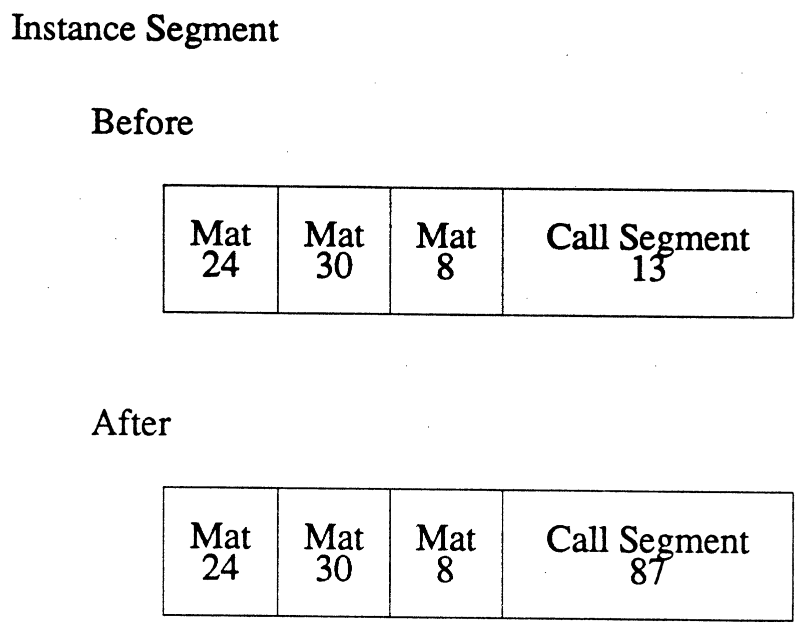


Figure 9-26 Dls_replace_instanced_obj

This routine deletes specified matrices from the matrix list of an existing instance segment as shown in Figure 9-28. The first and last arguments are used to specify which matrices in the segments matrix list are to be deleted. Both of these arguments are pointers into the list of disp_matdescr structures that is maintained by the modeling system for the instance display object for which this instance segment corresponds. If the first and last arguments are the same, then only one matrix is to be deleted from the segment.

The *devnm_dls_replace_instance_matdescr* declaration is:

```
void
devnm_dls_replace_instance_matdescr( instance, first, last, matdescrs )
dl_seg * instance;
disp_matdescr *first, *last;
disp_matdescr * matdescrs;
```

This is a combination of the deletion and insertion operations as shown in Figure 9-29.

The first and last arguments denote the matrices in the segments matrix list that are to be deleted. The matdescrs argument points to the first disp_matdescr in a list of disp_matdescrs that will be inserted into the segment's matrix list. The new matrices are inserted in the place of the ones deleted.

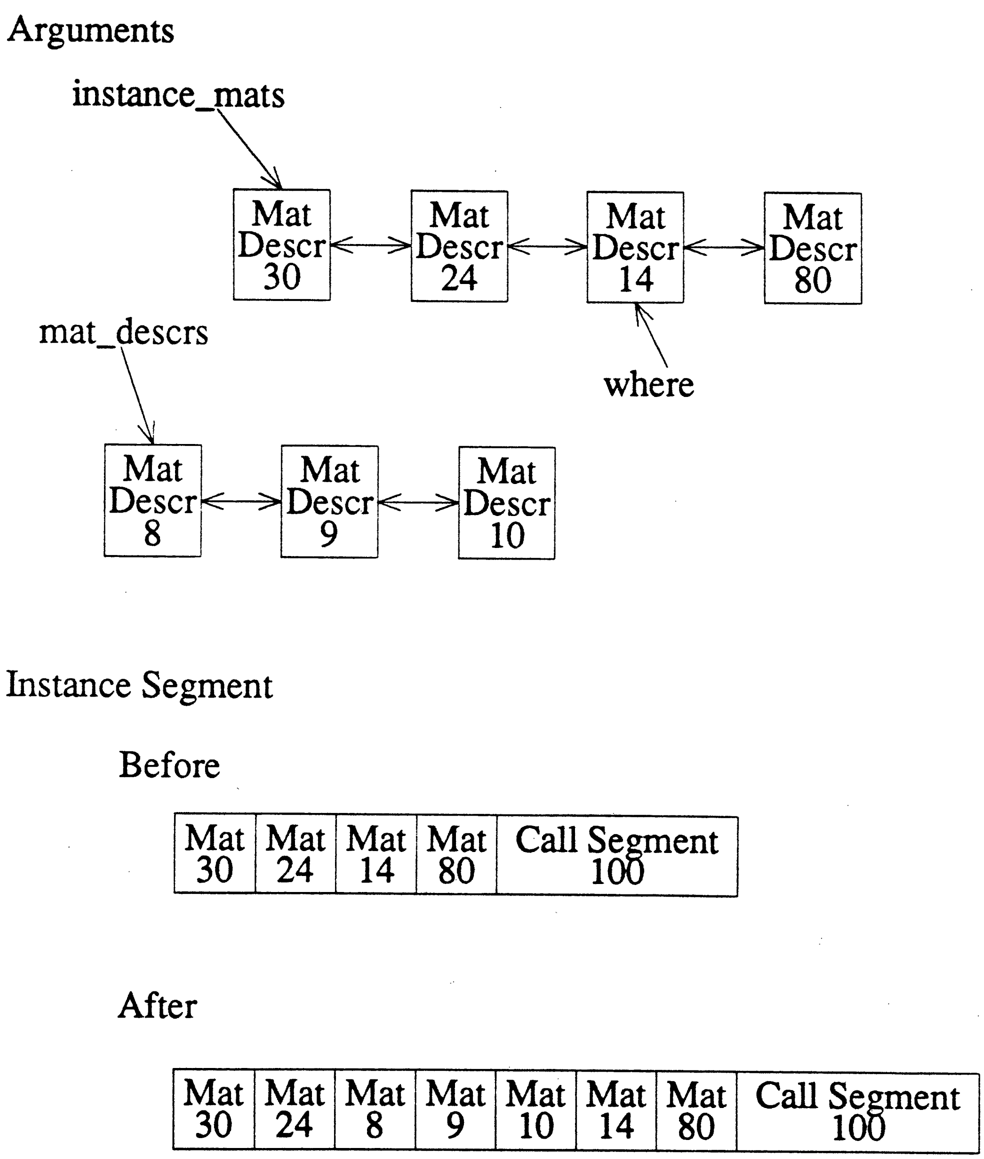


Figure 9-27 Dls_insert_instance_matdescr

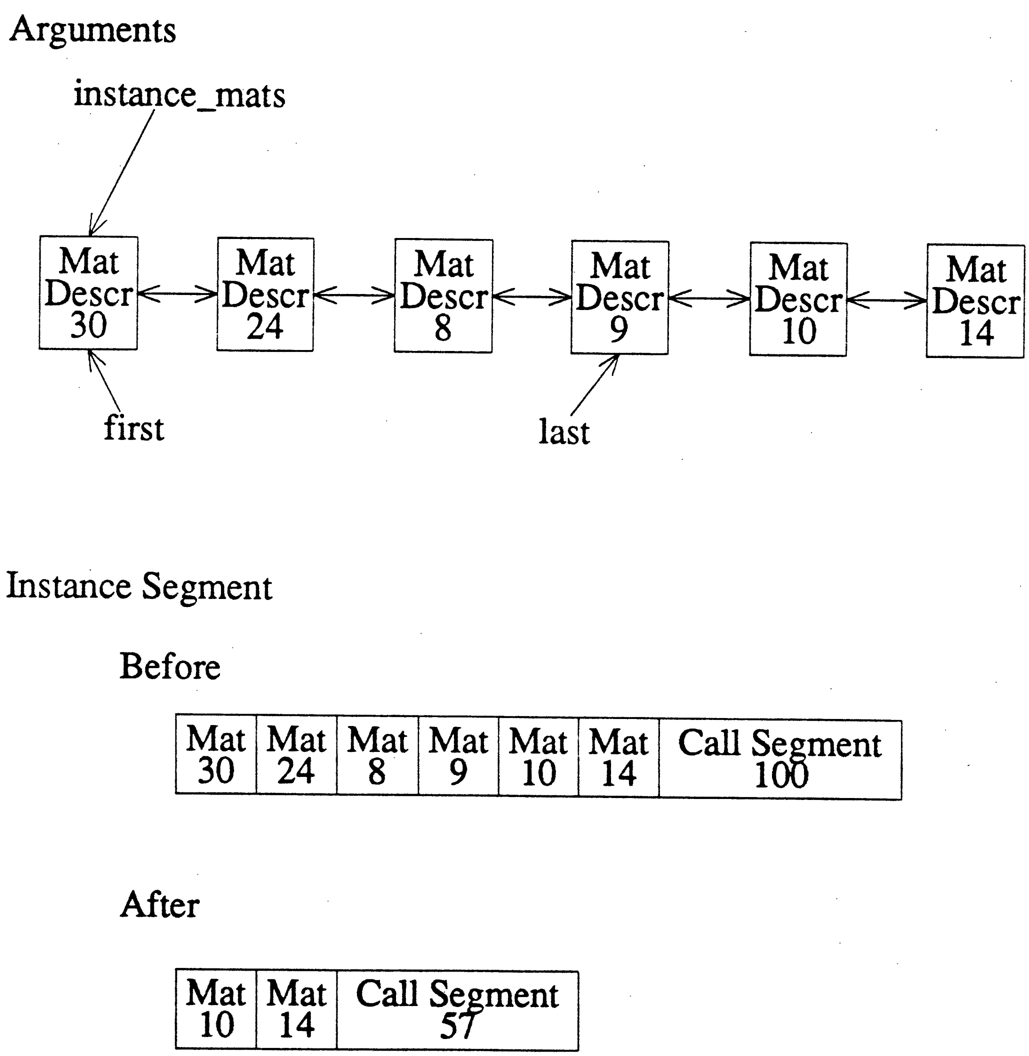


Figure 9-28 Dls_delete_instance_matdescr

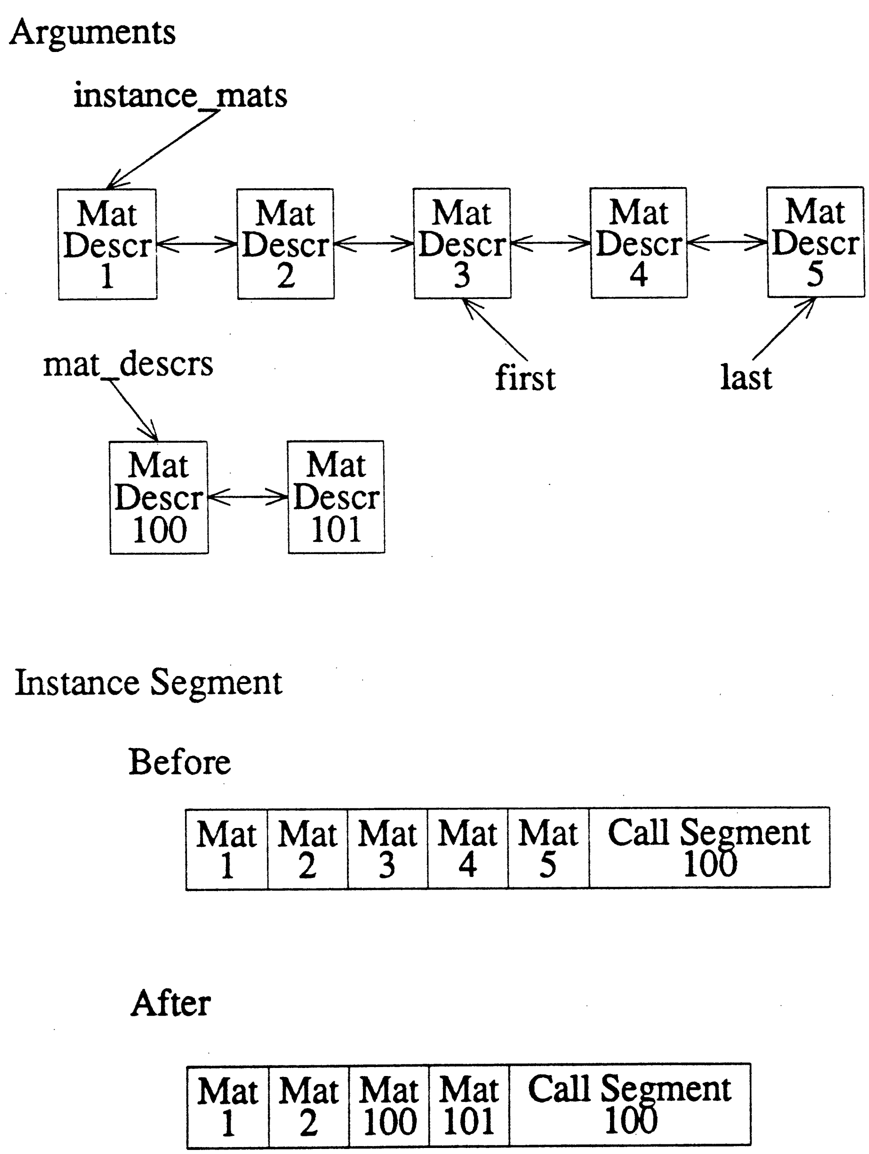


Figure 9-29 Dls_replace_instance_matdescr

If your device has a "replace" editing operation, then it should be used rather than combining the operations of deletion followed by insertion.

The *devnm_dls_modify_instance_matdescr* declaration is:

```
void
devnm_dls_modify_instance_matdescr( instance, which )
dl_seg * instance;
disp_matdescr * which;
```

This routine modifies a matrix in the matrix list of an instance segment. The which argument is used to specify which matrix in the segment's matrix list to modify. This argument is a pointer into the list of disp_ma

9.3 Building Support for a New Device

9.3.1 Device Requirements and Characterization

Before writing a new display module, you must clearly understand the capabilities and structure of the display device and the display operations which it will be expected to carry out under **shape_edit**. There are a series of detailed design decisions to be made as to exactly how the specified display operations will be mapped into actions the display device will perform.

Often, it will make sense to implement the display device module in stages, gaining experience with the device and correcting your idea of its capabilities. For example, it is recommended that you use the whole screen as a single output window at first and defer window management and interaction issues until after the basic display capability is implemented.

Segment Handling Requirements

Some of the major segment handling requirements of a display device were discussed at the beginning of the section on Modeler and Display Data Structure Correspondence (see section 9.1.3 [Modeler and Display Data Structure Correspondence], page 205). These are reviewed briefly here.

Device segments representing display primitives must be callable from device segments representing windows, groups, and instances. (And recall that groups and instances also are display primitives, so groups can contain instances, for example.) Instance objects must contain a sequence of transformation matrices which are concatenated and applied to the geometry.

The contents of a segment must be able to be replaced without changing the segment handle. Segment editing is required, but only for modifying the sequences of matrices and segment calls in group and instance segments.

Display List Versus Bitmap

The most fundamental issue affecting the display module design is whether the display device contains a segmented, hierarchical 3D display list and can manipulate it in the ways required by **shape_edit**.

"Bitmap" devices which have no display list at all are simple to figure out, since it is clear that the display list will be built and interpreted by C code. The display device operations you have to implement then concentrate on drawing vectors that have been transformed by C code. The device may have a hardware operation for drawing vectors, and may also clip those vectors to the window for you, in which case C code for those functions is unnecessary.

"Good" display list devices with segment calling and editing capability in their display list manager and strong matrix multiplication and clipping engines in their display list follower should also cause no problems. PHIGS standard display devices will all provide these capabilities, at least in software.

More difficult are "marginal" or "poor" display list devices that are missing capabilities. Some blend of simulated and real capabilities will result, and the display device module will have to maintain a simulated display list like the bitmap devices do **in parallel with** the real display list.

For example, if a display list device lacks the capability to multiply matrices, it will have to keep matrices in the simulated display list and traverse the simulated display list multiplying matrices and setting the matrix products in the real display list whenever a matrix is changed.

A display device on the GKS model without the ability to call segments within other segments also presents problems. It would be possible to maintain a full simulated display list and parallel device display list with flattened copies of the object hierarchy under each instance object, but it would be a lot of work.

Device Driver and Library

A special device driver is necessary to connect DMA devices into the Unix kernel. Ethernet devices communicate via TCP socket connections, eventually sending data through the Ethernet interface driver. Serial devices communicate via RS232 terminal lines connected to a serial port on the computer, sending data through the "tty driver". Note that serial lines do not support high performance display device communication, even at 9600 baud.

A "device library" is a set of C callable functions that pass commands and data to a display device. Often, there will be a good one supplied with the display device, in which case you need only understand clearly how to use it well. Other times, there will be a device driver but no library (or worse, an inefficient or restricted one), and it will be necessary to implement a library or construct raw device commands communicated through the Unix write, read, and ioctl functions. For command stream oriented devices, the "Standard I/O" fread, fwrite, and fflush functions will yield greater throughput because they buffer the I/O.

Window Manager Location

Although it is useful to start implementation with the device screen treated as a single window, **shape_edit** display devices must have windows and window management and viewing controlled by display device peripherals. The window management functions could reside in many places, as long as they produce the proper appearance on the screen:

- A workstation operating system.
- Downloaded code to a programmable display device.
- Device driver or library code managing a display list.
- Generic window management code managing a display list.

Local Viewing Operations

Smooth, dynamic motion is an important means of understanding the shape of a geometric model. It is important that the viewing position and direction be controlled by continuous positioning peripherals such as a mouse or tablet, or valuators such as knobs, forming a closed interaction loop involving the hand-eye coordination of the viewer.

Display devices need to be able to display different views at a frequency in excess of 20 times a second for this to be worthwhile, and the display should not "blink" or visibly clear the screen while showing a new view.

3-D dynamic viewing control is most likely going to have to be programmed as a local action of

the display device, since communications to **shape_edit** would slow down the viewing interaction too much.

Appearance

Depth cueing (modulating the intensity or color of a line in proportion to its transformed Z coordinate) is an important aid to visualizing three dimensional shapes, particularly in combination with smooth viewing motion. For **shape_edit**, it is probably preferable to support a small number of colors with depth cueing, rather than a large number of flat colors if you have the option.

Curve and Surface Display

The display device may be capable of local curve display, or less likely surface display. If so, it will be necessary to try it out and trade off B-spline display computation using the generic_draw functions vs. the display capabilities of the device. Often, devices with curve capabilities base them on 4x4 matrix multiplier in their display engine, which can have some pitfalls: they may be limited to cubic curves or not provide for [x,y,z,w] control points of rational B-splines.

It may also be necessary to translate Alpha_1 nonuniform knot vector, rational B-splines into uniform B-spline or piecewise polynomial form by subdivision before feeding them to the display device. It is also a good idea to subdivide at any multiple knots (of multiplicity greater than or equal to K - 1) even if the device knows about nonuniform, rational B-splines but evaluates them on even parametric increments, since doing so will miss corners (slope discontinuities) in curves that do not exactly fall on sample points.

9.3.2 Setup Script

There is a rather mechanical setup process necessary to initially hook a new display device module into **shape_edit**. The process is initiated by running the shell script

new-display

Output <...> ...

The script prompts you for a capitalized device name. Choose this name carefully, as it will be used to identify the directory for the device support as well as a number of constants, variables and routines within both the **shape_edit** and C support code. The identifier should not be too short, and should probably not be the name of the vendor of the device, since they will probably build another device or model sometime. If there are other variants of the device that somebody else may want to support, the device symbol should indicate the variant for which you are building support. For example, don't use 'EandS' or 'Ps' or 'PicSys' for an Evans & Sutherland display device. Use 'Ps300', 'Mps', or 'Ps2' instead. You may want to consult with an Alpha_1 staff member to choose an appropriate name.

The **new-display** script checks that the name you have chosen meets certain requirements, including not conflicting with any existing device names, then it asks you for a one-line description of the device, including the vendor and model. Usually a few words that describe the major device characteristics are also included — like whether it is a calligraphic or raster display.

New-display then prompts for the location of the device library.

Once all this information is collected, the script creates a subdirectory of $dlib and a working directory under it in which you will develop the device support. Several files which will be necessary for the development are generated on this directory, using the information you have provided in the script. These are

dev_config.h
: C include file with all device definitions, used especially by makefiles.

devnm.r
: Lisp definitions for making the new display device known to **shape_edit**. (Devnm is the identifier you gave for your device.)

devnm_fn_bodies.c
: An initial set of function bodies for the most basic display operations. You will fill these in, and add more operations to them to define the device support. (Again, "devnm" is the identifier you gave for your device.)

makefile.spec
: A specification for a makefile which keeps all your sources up to date.

On each device subdirectory, there is a file containing the bodies of some of the display manager functions for that device, for example the "iris_fn_bodies.c" file on the "$dlib/iris" subdirectory. These function bodies are delimited by

```
devnm_display_op: {
```

and

```
}
```

lines and automatically merged with proper calling sequences from the display declarations. All of the display operation declarations are contained in the file "$dli/display.str" which you may want to refer to while developing the device support.

You may choose to put functions in separate, normal ".c" files by putting a

    devnm_display_op:

line in the "devnm_fn_bodies.c" file (with no "{" on the line, and no function body following.) You will have to provide the proper calling sequences for separate functions and keep them in sync with "display.str" then.

There is a little code which is mandatory to make device and window control work the way **shape_edit** expects it to, and this code is included in the function bodies which are provided initially in your "devnm_fn_bodies.c" file.

The **new-display** script, after creating these files for you, generates a makefile from the "makefile.spec" file and runs it.

The makefile causes function bodies to be automatically merged into calling sequences from "display.str" by the **mk-device-module** script, producing a C file named "devnm_display.c". A warning will be given for any function bodies that do not match a display operation name. Any display operation calling sequences which do not match

    devnm_display_op:

lines in the "devnm_fn_bodies.c" file get a dummy body with the proper calling sequence so the function name will be defined for loading. For example:

```
/* TAG( iris_set_pickable_types ) */
void
iris_set_pickable_types( ntypes, typeset )
int ntypes, typeset[;

    /*...*/;
```

Do not edit the "devnm_display.c" file! The "devnm_fn_bodies.c" file is the one to develop. Note the calling sequences and comments in the "display.str" file carefully, since you are writing the bodies of functions with the same calling sequences as the display operation entries there.

Finally, the script generates and runs one small test program which contains all the framing for doing some real display, but actually does nothing. If this test program succeeds, you are ready to start adding device specific code. The operations and data structures you will need to deal with were described in the section on implementation design (see section 9.2 [Implementation Design], page 210). The next section provides a suggested order of implementation which allows you to do step-by-step testing of your device support as it is developed.

9.3.3 Test Suites in C

This section describes a series of test programs which are provided for developing new display support in small steps. Each suite of test programs adds to the functionality of the ones which came before and which are assumed to work. Using these test programs is the simplest way to debug your C code, before actually trying to load it into **shape_edit** and construct models. The programs can be generated on your working directory by saying, for example,

```
make s1_test1
```

The C code is tailored for your device from a template file, and the program compiled and linked. WARNING: These executables tend to take up a lot of disk space, so you may wish to delete them as you work through them.

The first test suite only tries to display disp_pts, the most basic object you will need: connected line segments. There are two test programs:

s1_test1
s1_test2

The first test program just puts up a disp_pts object on the display. No generic dispatching is be used in this program, instead the device specific routines are called explicitly by name. Generic dispatching is exercised in **s1_test2** and all the remaining test programs. For the first test you will need to implement:

devnm_open_dev
devnm_create_window
devnm_dls_dpts
devnm_dls_dgroup
devnm_dls_add_object_to_group
devnm_flush_dev
devnm_close_dev

The second test program, **s1_test2**, differs from **s1_test1** only in that generic dispatching is now done. You will need to implement:

devnm_select_dev

The second test suite completes the group operations:

s2_test1 add **devnm_dls_remove_object_from_group**

s2_test2 add **devnm_dls_dgroup**

The third test suite exercises polygon, point, and text display operations. They may be done in any order:

s3_test1	add **devnm_dls_poly**
s3_test2	add **devnm_dls_pt**
s3_test3	add **devnm_dls_text_str**

The fourth test suite adds curve and surface display operations. These can also be done in either order, but it's probably easier to do curves first:

s4_test1	add **devnm_dls_draw_mesh** and **devnm_dls_crv**
s4_test2	add **devnm_dls_draw_mesh** and **devnm_dls_spl**

The final test suite exercises all of the instance operations:

s5_test1	add **devnm_dls_dinstance**
s5_test2	add **devnm_dls_replace_instanced_obj**
s5_test3	add matdescr editing operations: **devnm_dls_insert_instance_matdescr** **devnm_dls_delete_instance_matdescr**
s5_test4	add **devnm_dls_modify_instance_matdescr**

Once the C test sequences work, you are probably ready to load the device module into **shape_edit**. Then you can use **shape_edit** to feed commands to your device module, which is more convenient than writing your own C test programs, compiling, loading, and running them to call your device module functions.

9.3.4 Testing from Shape_edit

You probably want to run a **dbx** between your emacs and your **shape_edit** for debugging. Find the

(setq rlisp-pgm "dbx ...shape_edit")

command in "$shd/tests.r" and use "C-^" (execute-mlisp-expr) to set the rlisp-pgm in your emacs. Then when you start an Rlisp process, **dbx** will be run automatically. The first time you hit "^C" or your C code runs into trouble, **dbx** will catch the exception and read in the symbol table information. The **dbx** continue command then passes the exception on to **shape_edit** so you can get a PSL backtrace if you want.

It is harder to make the symbol table information for all of the Alpha_1 code loaded, including your device module visible to **dbx**. One method for doing that is described below.

To load your device module into **shape_edit**, right after starting **shape_edit** give it commands:

```
cd ''$dlib/devnm'';
load oload;
oload( ''*.o -l $lib/libA1.a -lloc -l LIB_DEVNM -lm -lc'' );
on comp;
in ''devnm.r''$
```

You can customize these commands for your device by making a local copy of "tests.r". Substitute your device for "devnm" and the path to your device library for **LIB_DEVNM**. If you have no device library just delete the "-l LIB_DEVNM" part.

The **oload** process takes time. However, after you have done one successfully, as long as your C code (and **shape_edit**) don't change you can just repeat the oload from the cached ".oload.*" files by restarting the **shape_edit** and using the commands:

```
cd ''$dlib/devnm'';
load oload;
oload NIL;
on comp;
in ''devnm.r''$
```

This is faster.

In order for **dbx** to have information about the oloaded C code, you have to make a private **shape_edit** executable which has the symbol table information from the ".oload.out" file. To do this, after loading everything into **shape_edit**, say:

```
SaveSystem( Shape_EditBanner, ''my_shape_edit'',
        '((on break)(read-init-file ''shape_edit'')) )$
```

and start that **shape_edit** executable under **dbx** for debugging. Warning: **shape_edit** executables take about 3.5 megabytes as of this writing. Make sure there is sufficient disk space before attempting to save a **shape_edit** executable!

9.3.5 Installation

Once you have completed your **shape_edit** device module, it is time to install it in Alpha_1 so it will be included in the standard **shape_edit** executable that everyone runs. This is another good time to consult an Alpha_1 staff member for advice, before checking anything in. It is also highly recommended that you build a **shape_edit** from scratch on a working subdirectory before doing the checkin. Following this procedure should make building **shape_edit** fairly simple:

- First run make on your "$dlib/devnm/wd" working subdirectory to make sure that everything is up to date.
- You will need a working subdirectory of $shp that you can build **shape_edit** on, so

  ```
  cd $shp
  mknewdir
  cd wd       # Your working directory.
  ```
- Check out the "makefile.spec" onto the working directory and make the changes described below in the subsection entitled "Changes to $shp/makefile.spec".
- You will also want the **oload** phase of building **shape_edit** to load the ".o" files that drive your new device. So modify the

  ```
  #newDevOfiles = A1_SRC/lib/display/devnm/wd/ALL_OFILES
  ```

  line to uncomment it and change "devnm" and "wd" appropriately. You will want to delete any extraneous ".o" files from that directory before going on, like test suite program ".o" files. (It isn't too useful to link them all into **shape_edit**...) Remember to restore the "newDevOfiles" line to the way you found it before checking the "$shp/makefile.spec" file back in.
- In order for the makefile on $shp to be expanded correctly, you should make a symbolic link to your new "dev_config.h" file:

  ```
  ln -s $dlib/devnm/wd/dev_config.h $shp/wd
  ```
- Now generate the makefile:

  ```
  gen-makefile
  make local-oload
  ```

  The "local-oload" action works around a difficulty in the makefile expansion. Simply put, it causes a new set of C code to be prepared for loading into your working

directory copy of **shape_edit**, rather than using the prepared C code from the parent directory, $shp.

- You will need to go compile your "devnm.r" file into "devnm.b" on a working directory under $shd (See the section entitled "Changes to $shd" below.) Then make a link to your "devnm.b" file so it will be loaded into **shape_edit**.

```
ln -s $shd/wd/devnm.b $shp/wd
```

- Now run make, and if all goes well you should have a new **shape_edit** to try out. You can tell gemacs to start up the new one using:

```
M-X set rlisp-pgm ''$shp/wd/shape_edit''
```

  Be sure to

```
M-X kill-process rlisp
```

  first if your gemacs already has a **shape_edit** running underneath.

If you are able to grab your new device and show some data on it, you are ready to check everything in.

Checkin of the Device Support

These last few changes to system files insure that your device support is automatically kept up to date as changes are made to the system. They link your device support to makefiles that control building of Alpha_1 programs including **shape_edit**. You will need to check each of these files out before making the changes, and check them back in when done.

First, create the subdirectory of $dlib for your new display device at all of the other Alpha_1 installations:

```
cd $dlib
mkaldir devnm
```

Since the directory was already created by the **new-display** script, **mkaldir** will warn you:

```
src/lib/display/devnm/ already exists.
Continue? (y or n):
```

Your response should be:

```
y
```

You will also need to make a "debug" subdirectory of the new directory:

```
cd $dlib/devnm
mkaldir debug
```

(**Shape_edit** is currently loaded with the debug version of the display library, "libDL.a," which is constructed from ".o" files on the "$dlib/devnm/debug" directories. This allows debugging **shape_edit** under **dbx**.)

Check in your modified "dev_config.h" file to $ai and check in your new files to the device source directory: (Note that if you haven't checked out "$ai/dev_config.h" yet, you will have to do that and install your new device entry first.)

```
cd $dlib/devnm/wd
checkin $ai/dev_config.h
checkin makefile.spec devnm_fn_bodies.c
checkin ...anything else...
```

Changes to $dlib/makefile.spec

The "$dlib/makefile.spec" file gathers all of the display device code from the device subdirectories into one large library, "libDL.a". You will need to add an entry in the style of the ones already there:

```
%ifdef DEV_DEVNM
    devnm_SUBDIR = $(HOME)devnm$(SUBDIR)
    devnm_OBJS = $(devnm_SUBDIR)/objs
%endif
```

You must also add references to the $(devnm_SUBDIR) and $(devnm_OBJS) variables to the

```
SUBDIRS =
```

and

```
OBJS =
```

lines respectively.

Changes to $shd

Move the "dev.r" file under $shd where it is expected by **shape_edit**.

```
cd $shd/wd
mv $dlib/devnm/wd/devnm.r .
```

The "$shd/makefile.spec" file controls the Rlisp components of the **shape_edit** display manager, so you have to tell it about the "devnm.r" file. Add an entry to the "LFILES devices" list, of the form:

```
%ifdef DEV_DEVNM
    devnm.r
%endif
```

Now you are set to compile a debug version of "devnm.r" into "devnm.b:"

```
gen-makefile
make depend
make devnm.b
```

Finally, check in your changes.

```
cd $shd/wd
checkin makefile.spec devnm.r
```

Changes to $shp/makefile.spec

The "$shp/makefile.spec" file controls loading everything into **shape_edit**. Add an entry to the "Display device libraries" section of the "$shp/makefile.spec" file:

```
%ifdef DEV_DEVNM
    devnmLIB = LIB_DEVNM
    devnmARG = -l LIB_DEVNM
%endif
```

You must also add references to the $(devnmLIB) and $(devnmARG) variables to the

```
DEVLIBS =
```

and

```
DEVARGS =
```

lines respectively.

“Global Make” Support

The “$amk/srcdirs” file controls the “global make” of Alpha_1. Install an entry like:

```
$alpha1/src/cmd/shape/disp/devnm         default       staffa1
    $alpha1/src/cmd/shape/disp/devnm/debug       default       staffa1
```

after the entry for the “shape/disp” directory, following the style of the others you see there.

You will also need to check in “.normal_make” files to both the main device directory and the debug subdirectory. They suppress notification messages from **makeall** when nothing is really being done by **make**. Currently, all of the display device directory “.normal_make” files are identical, containing:

```
make -k makefile
'makefile' is up to date.
make -k depend
'depend' is up to date.
make -k default
```

9.4 Low-Level Design

This section describes implementation details at the **shape_edit** level and implementation of the communications between PSL and C. You should not need to understand (or even read) this section if you are just trying to provide support for a new device under **shape_edit**. If you are trying to extend the interface in any way, then you will need this information.

9.4.1 Shape_edit Details

Shape_edit DisplayDev structures store only a little information about the display device state. All the implementor specifies is is a device name string for error messages, and the name of a function that makes the device the current one for DisplayOp dispatching purposes at the C level. The display manager accumulates an association list of window IDs and window objects for windows on the device. A list of the objects which have display list segments on the device is also accumulated so the display list storage on the device can be restored if it is dropped and grabbed again.

DisplayDev objects are keyed by the device ID and kept in an association list in the global **Devices!*** variable.

Shape_edit DisplayWindow objects merely contain a C_Address slot which is a handle on a C window_object in the malloc heap, and a reference to a **shape_edit** group object which contains references to objects visible in the window. (See section 9.4.2 [Communication with C Level], page 248 for more details on what it means to have a C_Address in a **shape_edit** object structure.)

The **shape_edit** ModelObject type is actually a flavor inherited by all object types defined using **defModelObject**. Besides linkage slots that bind objects together with a graph of dependencies and support change propagation, there is a DisplaySegs slot that contains references to segments on each of the display devices where an object is visible. The DisplaySegs slot contains an association list of IDs of display devices as keys and C addresses of dl_seg objects as values. Figures 9-30 and 9-31 illustrate the **shape_edit** data structures and their correspondence with C display support and device segments.

“Display-df.r” contains the global state variables of the display manager package: (Those with “!*” at the beginning are boolean mode flags, set by “on” and “off”. The others have a variety of value

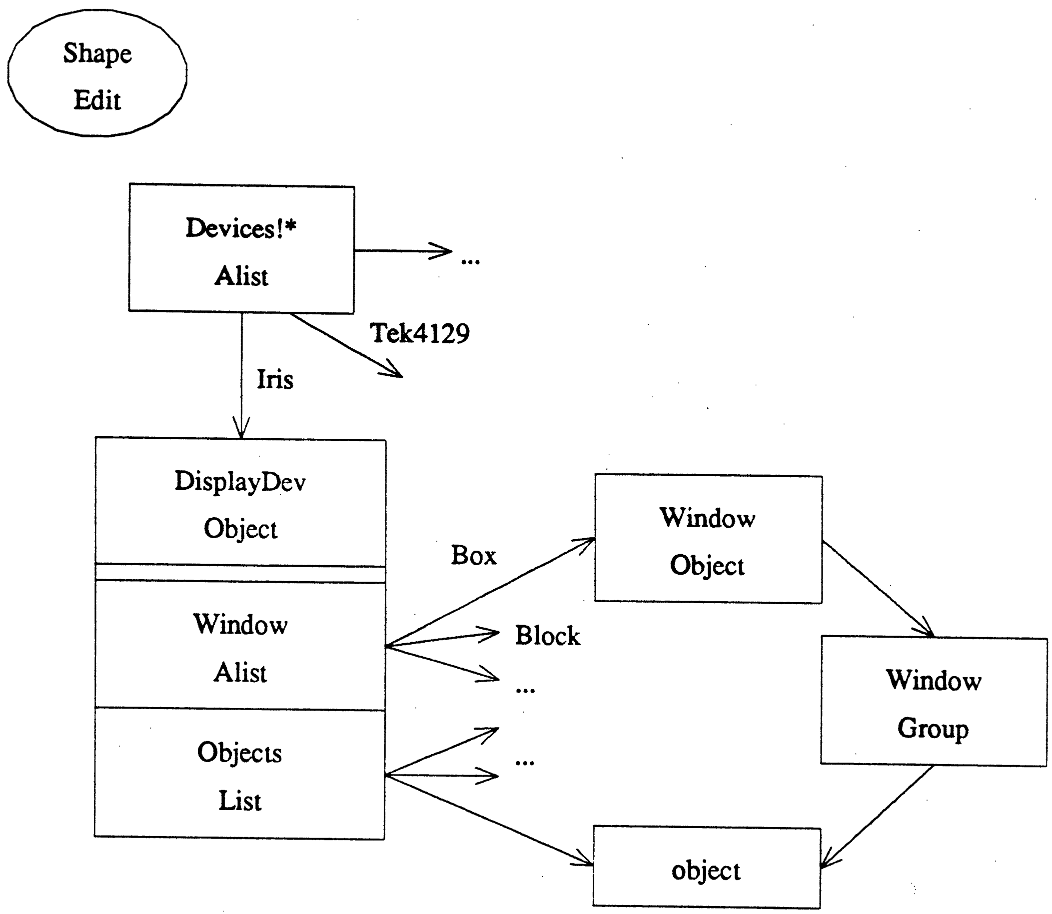


Figure 9-30 Shape_edit Display Manager Data Structures

types.)

- **Devices!*** contains the state of the display manager, in the form of an aList associating device IDs with DisplayDev objects.
- **CurrentDeviceName!*** contains the name of the current display device. The only effect of this is to specify a default device for the window create, remove, etc. functions if no other device is given. It is set by grabbing a device.
- **SelectedWindows!*** identifies the set of windows where newly shown objects are to appear, as a list of DeviceName . WindowName pairs.
- **!*MultiWindow** means that a new window **becomes** the selected window set if off. New windows are **added** to the selected window set if on.
- **!&ActiveDeviceName** is an internal variable to remember the device whose C Dis-

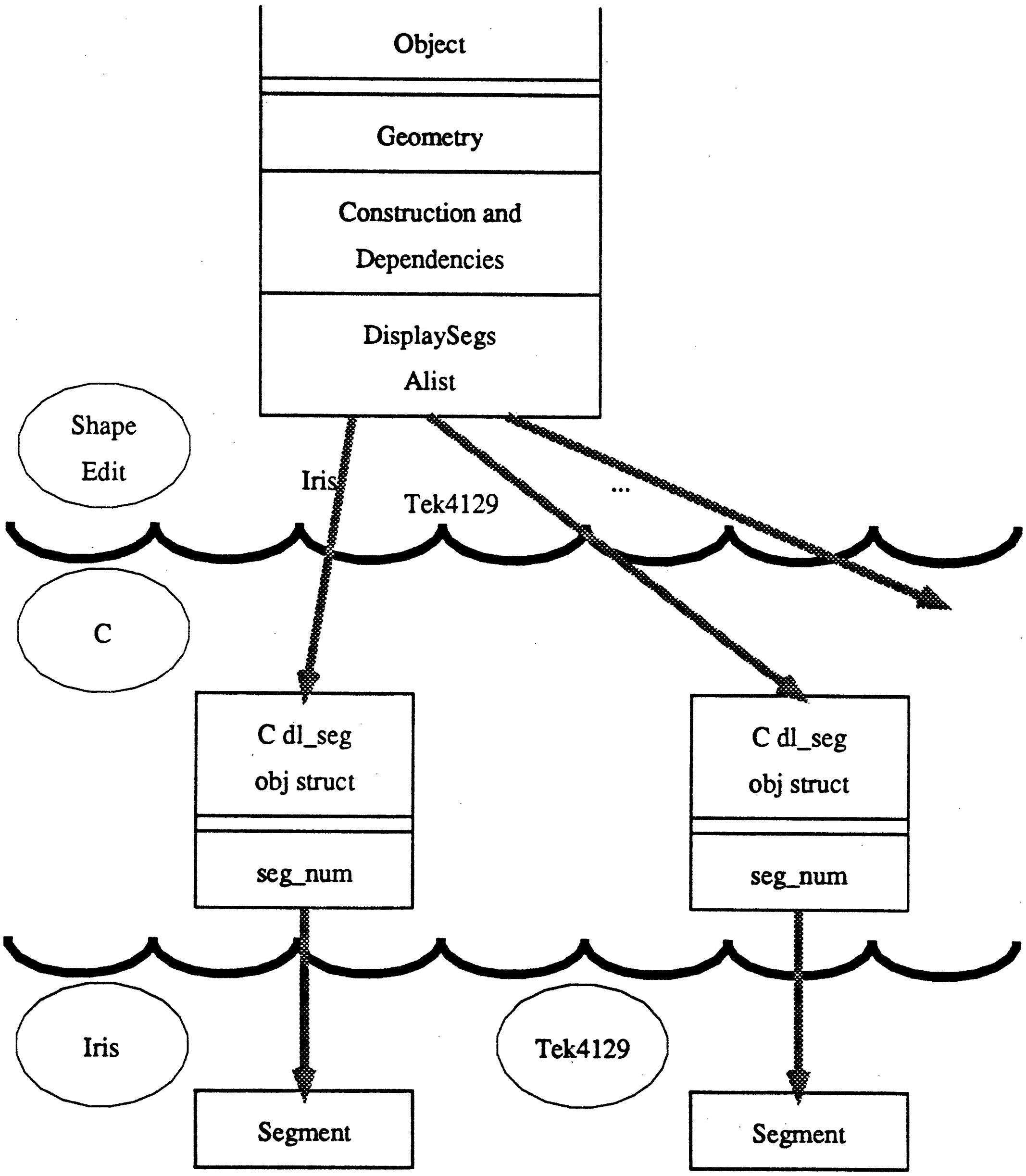


Figure 9-31 Shape_edit Display Device Data Structures (continued)

playOp functions will currently be called. Set by **!&cSelectDevice**. Used by DlsToC methods for structured objects such as groups.

9.4.2 Communication with C Level

Although Lisp and C are worlds apart in their data structuring approaches and storage management disciplines, we have implemented vaguely similar object-oriented programming environments in both languages. The decision was made to concentrate knowledge of the interconnections of model objects in the Lisp object forms, and concentrate numerical processing of B-splines and the calling of C display device library functions in C code.

The PSL to C interface translates Lisp model objects in the PSL heap into C equivalent objects in the malloc heap, calls a C function with the translated objects as arguments, and frees the C objects in the malloc heap after the C function returns. The translation is packaged as a generic **ToC** operation (method) defined for appropriate model object types. This process is shown for the display list segment creation operation in Figure 9-32. A back-translation process (**FromC**) for the return values of object-valued C functions is not yet implemented since it was not needed to implement the display manager.

The PSL to C interface overhead is kept down by the fact that bulky data for objects like B-splines and polylines is kept in a packed array form which is binary compatible with the C array data structures we have implemented. Thus only a block copy from the PSL heap to the malloc heap is needed in this case.

For display, some object types are translated into specialized C object types local to the display manager by **DlsToC** methods rather than the general C form produced by **ToC** methods. For example, a group model object is translated into a variable-sized block of segment handles, rather than a list of C geometric objects.

The remainder of this section presents more detailed information useful in understanding how C functions are called by the Rlisp code at the top level of the display manager.

C_Address slots of objects are used to keep track of "permanent" C structures which persist in the malloc heap and hold state for the C level of the display manager. ("Ephemeral" objects are constructed as arguments of C functions and freed immediately after returning to the PSL level.) C_Address slots contain numbers (or Nil) as far as PSL can tell. Those numbers are actually pointers produced by direct or indirect calls on the Alpha_1 C new function, which calls the malloc library function and returns a pointer to the allocated block. The C_Address numbers become interpreted as pointers to C objects in the malloc heap again by context when they are passed as arguments to C functions called by compiled PSL code.

C Function names within PSL are referenced with a leading underscore, e.g., the C new function is called as _new. This is due to the fact that the PSL **oload** utility loads C symbols by their true names as known to Unix, which have a leading underscore. This was done intentionally, so that PSL "pass-through" functions could be declared which transparently "wrap" C functions in a layer of argument translation and then call the C function.

PSL functions which call oloaded C functions must be run compiled, since the PSL interpreter does not know how to call foreign functions. A definition should be present in the source code that flags the underscore prefixed true name of the C function with the ForeignFunction property to inform the compiler to use a different calling sequence convention from that used in calling native PSL functions.

The current limitations in the calling of C "foreign functions" by PSL that should be kept in mind: no more than 5 argument expressions can be handled in a call and arguments which are structures

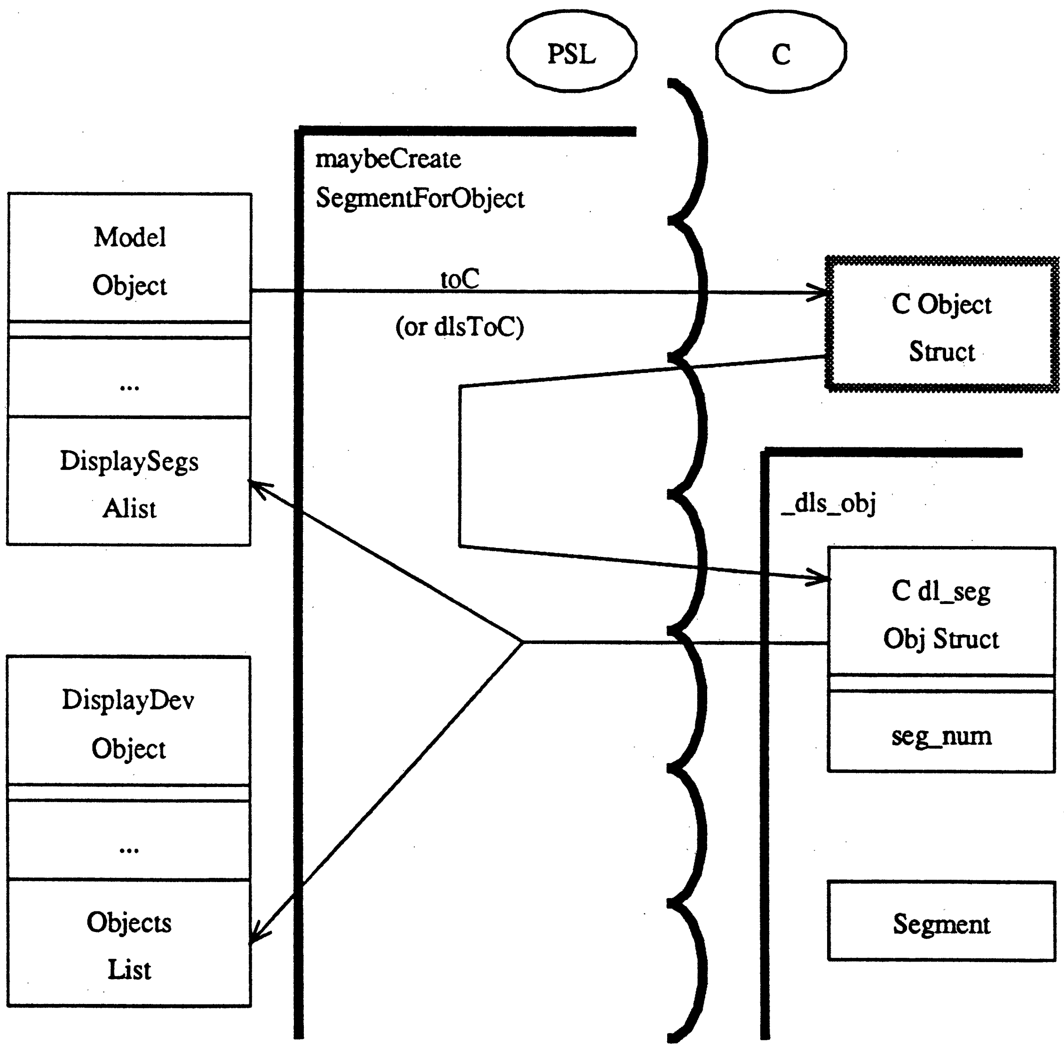


Figure 9-32 PSL to C Display Data Structure Creation

(rather than pointers to structures) cannot be accommodated.

ToC

There is some overlap between the data types of PSL and those of C that allows direct argument and return value passing while calling C code from PSL. In other cases, type conversion functions have been implemented in file "to_c.r," and sometimes a specialized C intermediate function is required as well.

Integer (and pointer) arguments and return values pass straight into C code and back from "normal" PSL code.

Boolean flags require a little bit of translation from the PSL "Nil or not" form to the C "0 or 1" form, provided by the **booleanToC** and **booleanFromC** functions.

PSL floats correspond to the C double type. The PSL compiler assumes a single size for all arguments passed on the stack to C functions, so a float is passed as two arguments (really!) generated by the functions **floatArg1** and **floatArg2**. (These functions take care of coercing a PSL generic number argument to floating point if necessary, so be aware that a garbage collection may result when they are called with integer arguments.) There is currently no way of passing a floating point return value directly back from a C function.

Strings are stored in PSL with both a length field in their header and a trailing null byte terminator as needed by C. The function **stringPtr** takes two arguments: a string object and a function ID which is printed in the error message that results if the first argument is not a string. It returns a C pointer to the body of a PSL string object, which will be wrong if the string is in the heap and a PSL garbage collection relocates the string so the pointer no longer points to it.

This is important: You must do nothing that allocates space from the heap between the time you get a C pointer into an object in the heap and the time it is used, because someday that allocation is going to cause a garbage collection and your code will fail randomly. This applies to all types of pointers, not just pointers to strings. One safe way to write the code is to do all construction of PSL objects beforehand, putting the results in local variables in your function, then get pointers into PSL objects referred to by the local variables just before passing the pointers as arguments to the C function call.

Another safe thing to do is to use space in the PSL BPS or WArray spaces or the malloc heap, which are not garbage collected so pointers into them remain valid. In particular, print strings of PSL IDs (returned by **id2String**) are in BPS space. The **exportString** function copies any string into C malloc space and returns a pointer to the C string.

Arrays are passed by address in C, so a PSL list or vector structure is packed into a word vector in PSL, and a pointer to it is passed to C code. The **wordsPtr** function returns a pointer to the body of a word vector which is its first argument. Its second argument is a function ID to print in error messages.

PSL "X-vector" operations, including subscript operations, apply to word vectors and are a convenient way to fill in word vector objects. It is probably fair to depend on the fact that the PSL "word" is equivalent in size to a C "long", "int", or pointer, so word vectors can be constructed and passed as arrays of those types.

The caveat about garbage collection for strings above applies to word vectors as well, unless they are in WArray space. The **staticWords** function is given the number of words to allocate and returns a word vector whose body is in WArray space. This has two advantages: it is immune to relocation if garbage collection occurs, and it can be reused without creating garbage in the heap. The disadvantage of course is that the space is of a fixed size, and once allocated the space is used up for the duration of the PSL session.

The **exportWords** function copies a word vector given as its first argument into C malloc space, returning a C pointer to the copy. Its second argument specifies the index of the first element of the vector to copy. Use 0 to start at the beginning of the vector. The third argument, if Nil, specifies that the rest of the word vector is to be copied. If the second argument is not Nil, it tells how many words to copy.

It is probably not a good idea in general to construct C structs in PSL word vectors and then pass them to C. That can result in nonportable code because of the differences in C structure element alignment from machine to machine. The Alpha_1 "_mk_..." functions take structure elements as arguments, allocate and fill in the structure of an object in the malloc heap, and return a pointer to the object. It is of course necessary to use the dispose or "_fr_obj" functions to release the space after use.

However, the PSL to C interface depends on the fact that the simpler Alpha_1 geometry struct types, which must be passed as arguments to C functions, are quite array-like and shouldn't have portability problems from alignment. This includes the "point", "matrix", "curve", and "surface" types which are packed into PSL word vectors for reference by C code.

The FloatArray package constructs packed arrays as word vectors with some int sizes in leading words and two words per float element in the array body, corresponding to the "matrix" structure declared in the "$ai/matrix.h" file.

Shape_edit point objects are packed into word vectors by the **packCoords** function, as shown in Figure 9-33. The first argument is a destination vector, which is packed with the floating point coordinates of the point object and returned. Remember that there must be at least twice as many words as coordinates, since floating point numbers are double precision in PSL. Nil may be given as a destination to cause a new destination vector to be allocated in the heap. T specifies that a single, static internal buffer will be used. The second argument is the point object to be converted, and the return value is the word object where the coordinates were packed.

The **packCurve** and **packSurface** functions take destination arguments like **packCoords** above, but return C curve and surface pointers respectively. A diagram of the process of sending a curve from PSL to C is shown in Figure 9-34.

ToC methods are implemented for the **shape_edit** text, point (e2Pt, e3Pt, p2Pt, p3Pt), curve, and surface object types so far. **DlsToC** methods are implemented for polyline, group, and instance object types.

The few "specialized C intermediate functions" mentioned above are in file "misc_interface.c". The "mk_...2" functions call the corresponding "mk_..." functions which demand a structure argument on the stack, and are called with a **pointer to** a structure, which is loaded on the stack and passed on.

One problem with the C generic operations as currently implemented in the C "object management" is that the generic dispatchers are C **macros** declared in the (automatically generated) file "$ai/operations.h". The "c-operations.r" file contains equivalent (manually generated) dispatchers which call the appropriate dispatch function for an operation, passing it the "type tag" of the C object. (t_tag is implemented in file "misc_interface.c".)

Now that we have gotten from the Rlisp **shape_edit** level and down through the PSL to C interface code, we are almost to the domain of "normal" C code which defines the specific display device operation (**DisplayOp**) functions for each display device. But first, this section describes a bunch of code generated from a master specification file, "display.str", including the "generic DisplayOp" functions.

Generic DisplayOp functions are actually dispatchers from a generic name to a specific function for

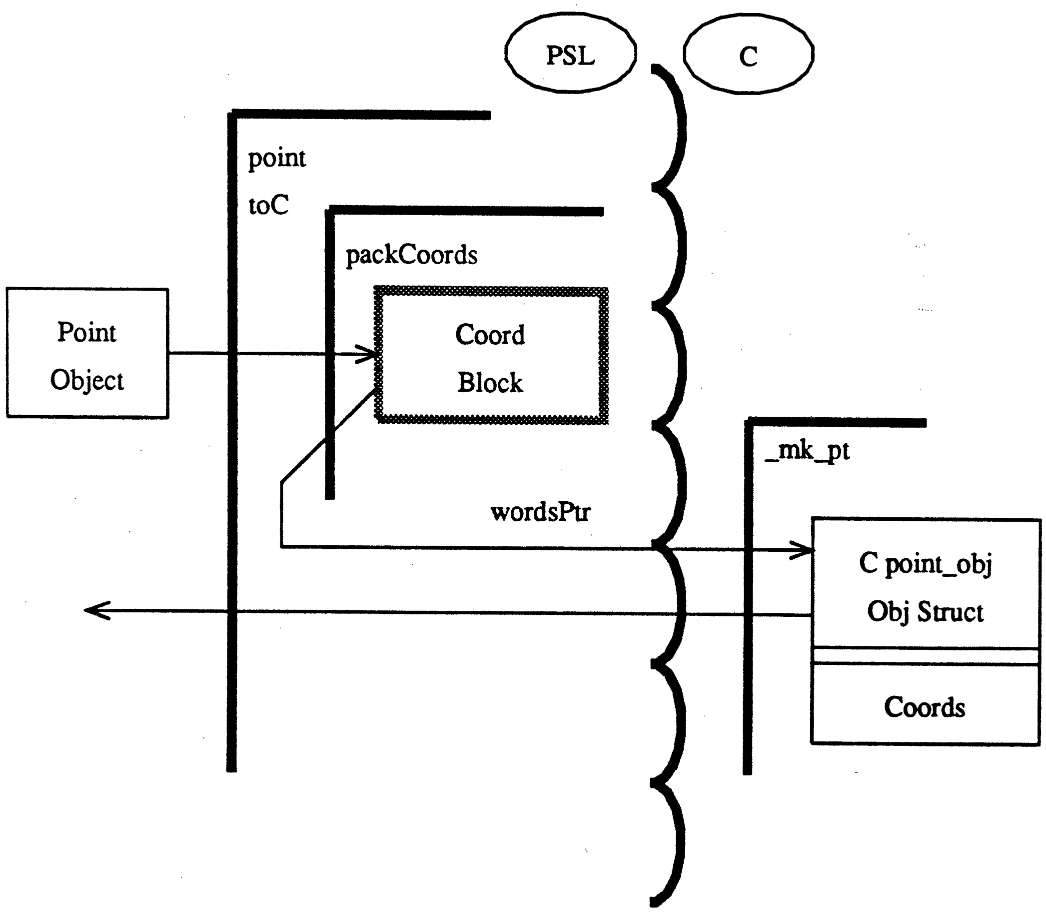


Figure 9-33 Point toC Operation

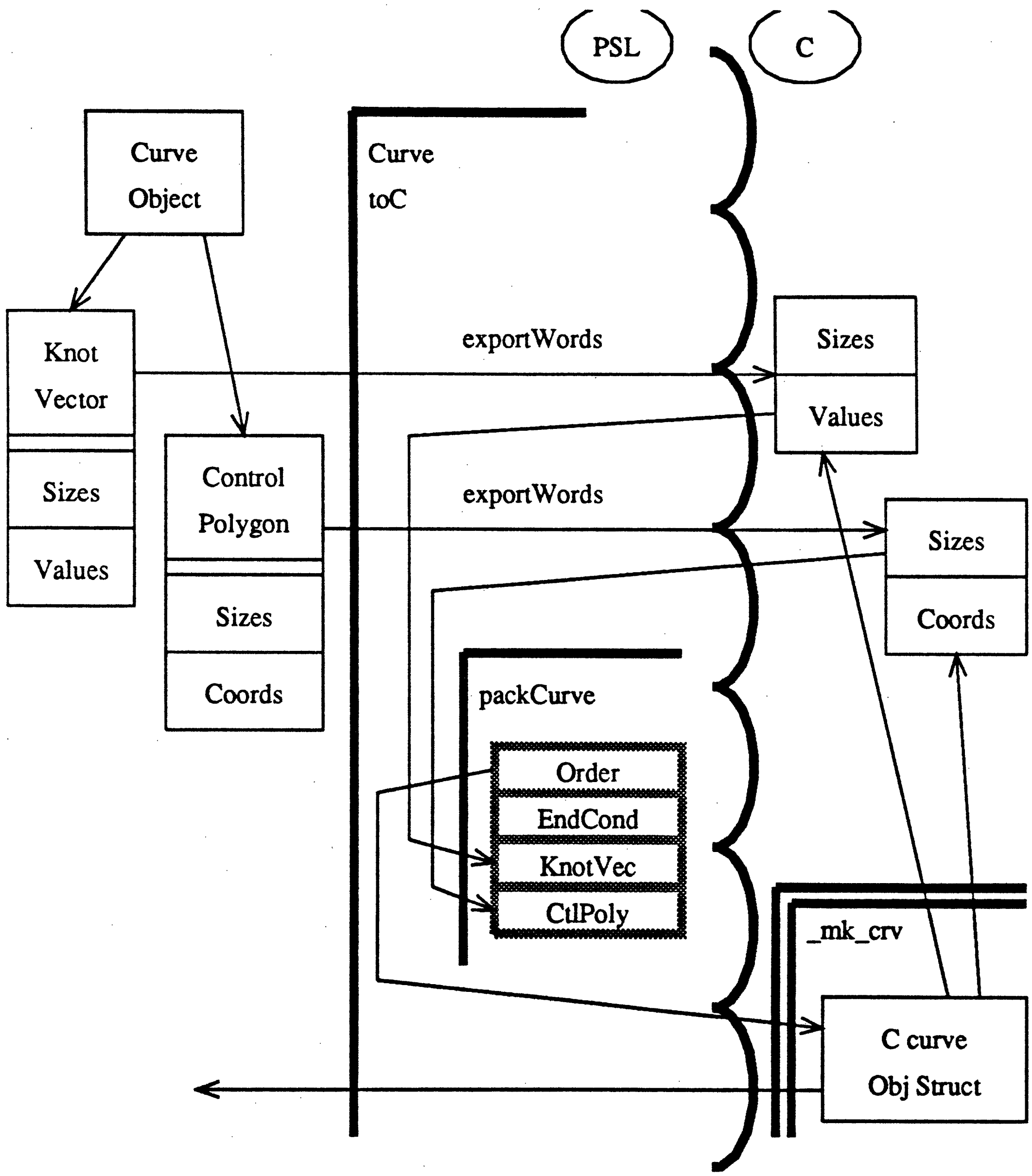


Figure 9-34 Curve toC Operation

each display device. The generic name is the one declared in the "display.str" file, and the specific names are all the same except for a device name and underscore prefix. Thus "dls_crv" is a generic DisplayOp name and "iris_dls_crv" is a specific DisplayOp name.

Note that this is a different flavor of dispatching from the "generic operations" generated by the Alpha_1 object management utilities for protocol operations. Object management dispatchers choose a function based on the type tag of their first argument, so "dls_obj" will call "dls_crv" for a "curve_object". DisplayOp dispatchers choose a function based on the curr_device variable, so "dls_crv" will call "iris_dls_crv" if the iris is the selected device.

The **shape_edit** level of the display manager transparently handles the details of managing multiple devices, so the C level needs only to dispatch to the DisplayOp functions of the single display device selected at any time. The support module for each display device should be unaware that there are any other display devices or dispatching between them.

The mechanics of DisplayOp dispatchers involve a series of pointers starting at the **curr_device** variable. **curr_device** is set to point to a display_dev structure by the device **select_dev** function. The display_dev structure contains a dev_fns slot, which points to a dev_dispatch structure. DisplayOp dispatchers make indirect calls on functions pointed to by slots of the dev_dispatch structure of the current device.

For example, the DisplayOp dispatcher for "dls_crv" is:

```
/* TAG( dls_crv ) */
dl_seg *
dls_crv( crv, seg )
crv_object * crv;
dl_seg * seg;

    return (* curr_device->dev_fns->dls_crv) ( crv, seg );
```

All of the DisplayOp dispatchers are in file "display_dispatch.c", which is generated from "display.str" by the **mk-display_dispatch** script.

The dev_dispatch struct contains a complete list of DisplayOps as its slots. The definition of dev_dispatch is in file "dev_dispatch.h", which is "#included" in "display.h" and generated from "display.str" by the **mk-dev_dispatch** script.

The final question is how pointers used in dispatching get initialized to contain the right values. On each device subdirectory, the **mk-device-dispatch** script generates a file like the iris_dispatch.c file on the "$dlib/iris" subdirectory. It contains declarations of all of the "iris_" prefixed DisplayOps functions, and initializes an "iris_dispatch" dev_dispatch structure to contain pointers to all of them. All that is needed to complete the chain is for the **iris_select_dev** function to set the "iris.dev_fns" slot to point to iris_dispatch.

9.5 Unfinished Areas

This section describes the not-yet-implemented or in some cases not-yet-designed parts of the **shape_edit** display manager. It will eventually go away, as we finish designing and implementing.

Bounding Box and Autoscale

The bb_obj operations will be available to decide what scale to use for curve and surface fineness. The code to use them should be stolen from that in the **view** program, just as the rest of the curve

and surface display code was.

Issues to resolve include when to run the geometry in a window through the autoscale function. Initially it should probably be done only when an autoscale function is called explicitly.

Similarly, the autoscale information could be used to autocenter the viewing transformation on a window when desired.

Display List Window Support

A generic window manager will have to be written for use on display list devices that do not provide a window manager. It would basically treat the display list on the device as a group of window segments, where a window frame and group of objects would be visible in each segment. All of the window position/reshaping, pushing/popping/iconing operations will have to be provided under control of a positioning device.

Bitmap Class Devices

Generic simulated display list support for bitmap devices is being worked on. Such devices would basically subscribe to operations in bitmap.str instead of "display.str", with the simulated display list functions filling in the DisplayOp functions and calling "bitmap.str" functions. BitmapOp functions will at least involve window control and linesegment, point marker and text drawing.

Unimplemented Geometric Object Types

Polygon objects exist in the C code in a linked list form, where a polygon has a list of oriented contours which can be the boundaries of either the "outer edge" of a polygon or "inner" (hole) edges. Contours are circular lists of vertex objects, which can have shade and/or normal attributes.

Initially, **shape_edit** does not have polygon objects, nor obviously a **toC** operation that can construct the C polygon objects. Eventually, we should presumably construct polygon objects equivalent to the C code. Precisely what they would be used for in modeling, and how they should be handled on the display, are open questions.

There are also a set of primitive geometry objects in **shape_edit** which do not as yet have an equivalent in C. These include line, arc, circle, and plane objects. Presumably C equivalents should be defined so as to have a means of dumping models into ".a1" or ".bin" files and loading them again. The initial display of lines puts out polylines, and arcs and circles are displayed as curves. Planes will probably give an error...

User Interface Support

The decision was made to first support the "programmer's user interface", get a good set of display devices working, and then work on generalizing the PS300 prototype "screen and menu user interface" to work with the new **shape_edit** display manager. Thus, the specifications for menus and input events in "display.str" are science fiction at this point.

Index

D